Dynamic, Interactive Virtual Environments

Kristopher James Blom
MIN Fakultät, Dept. Informatik
Universität Hamburg

zur Erlangung des Akademisches Grades
submitted in partial fulfillment of the degree of
Doctor der Naturwissenschaft (Dr. rer. nat.)

September 2008

Abstract

Virtual Environments enable the creation of fantastical worlds, limited only by the imagination of their authors. They can come to life through specialized software, constrained only by the prowess of the developer. Unfortunately, the level of sophistication required to make statically modelled Virtual Environments (VEs) come to life is a hurdle that makes such environments uncommon. In contrast, the ability to create such environments is widely desired and recognized as necessary. The success and effectiveness of many application areas hinge on the creation of interesting, captivating, affective VEs. This work focuses on such environments and the tools necessary to create them.

The first portion of the research presented investigates such experiential environments. Based on the idea that dynamic and interactive components are what make them interesting and exciting, an exploration and analysis of the design space of **Dynamic, Interactive Virtual Environments** (DIVEs) is performed. Three main content types are defined and their design spaces explored: Dynamics, Dynamic Interaction, and Interactive Dynamics. The analysis of these components provides insights into the potentials of DIVEs and provides new understanding to what is truly required to create them.

Based on requirements and use cases identified in the investigation of the DIVE design space, a framework of support for the development of DIVEs is proposed. The *Functional Reactive Virtual Reality* (FRVR) framework developed supports DIVE creation in Virtual Reality contexts. FRVR is built up from the combination of the emerging programming paradigm Functional Reactive Programming (FRP) and existing Virtual Reality (VR) software. The FRP paradigm was selected as a basis system, because it is well suited to the hybrid nature of DIVEs and VR. Extensions to the existing functionalities of FRP create a system that meets the requirements for DIVEs and provides many new and advanced functionalities that previous VR systems have lacked. Example applications demonstate how DIVEs can be implemented using FRVR and testify to the flexibility and power of the approach.

Kurzfassung

Virtuelle Umgebungen ermöglichen die Erstellung vielseitiger Welten, die nur durch die Vorstellungskraft ihrer Autoren beschränkt sind. Diese Welten können durch spezialisierte Software kreiert werden, welche wiederum nur durch das Können der Entwickler limitiert ist. Bedauerlicherweise stellt das Schwierigkeitsniveau, das benötigt wird um statisch modellierte Virtuelle Umgebungen (Englisch: Virtual Environments, VE) lebendig zu gestalten, eine Hürde dar, die solche Umgebungen zu einer Seltenheit macht. Trotzdem wird die Möglichkeiten solche Umgebungen zu schaffen oft gewünscht und allgemein als notwendig angesehen. Der Erfolg und die Effektivität vieler Anwendungsgebiete hängen entscheidend von der Erzeugung interessanter, fesselnder und affektiver Virtueller Umgebungen ab. Diese Arbeit konzentriert sich auf solche Umgebungen und auf Werkzeuge, die notwendig sind um diese zu erstellen.

Der erste Teil der vorgestellten Forschungsarbeit untersucht solche erfahrungsreichen Umgebungen. Basierend auf der These, dass es dynamische und interaktive Komponenten sind, die diese Umgebungen interessant machen, wird eine Erkundung und Analyse des Entwurfsraumes von Dynamischen Interaktiven Virtuellen Umgebungen (Englisch: Dynamic, Interactive Virtual Environments, DIVE) durchgeführt. Dazu gehören Dynamik, dynamische Interaktion und interaktive Dynamik. Die Analyse dieser Komponenten eröffnet Einblicke in das Potenzial von DIVE und gibt ein neues Verständnis darüber, was benötigt werden um sie zu kreieren.

Basierend auf den Anforderungen und Anwendungsfällen, die in der Untersuchung des DIVE Entwurfsraumes identifiziert wurden, wird ein Rahmenwerk vorgestellt. Das Functional Reactive Virtual Reality (FRVR) Rahmenwerk unterstützt die Erzeugung von DIVEs im Kontext von Virtuellen Realität (VR). FRVR ist eine Kombination des neuen Programmierparadigmas Functional Reactive Programming (FRP) und existierender VR Software. Das FRP Paradigma wurde als Basis ausgewählt, weil es sehr gut zur hybriden Natur von DIVEs und Virtueller Realität passt. Erweiterungen der existierenden Funktionalitäten von FRP erzeugen ein System, das den Anforderungen von DIVEs entspricht und viele neue und fortgeschrittene Funktionalitäten zur Verfügung stellt, die die bisherige VR Systeme nicht aufweisen konnten. Beispielanwendungen demonstrieren, wie DIVEs mit FRVR implementiert werden können und verdeutlichen die Flexibilität und Mächtigkeit dieses Ansatzes.

Declaration

I affirm that my dissertation entitled "Dynamic, Interactive Virtual Environments" is my own work excepting where otherwise acknowledged and referenced in the text. I hereby declare that this thesis has not already been accepted in substance for any degree and is not being currently submitted in candidature for any other degree.

Eidestattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Dissertation mit dem Titel "Dynamic, Interactive Virtual Environments" selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe. Diese Dissertation habe ich bisher weder im Inland noch im Ausland in irgendeiner Form als Prüfungsarbeit vorgelegt.

Contents

List of Figures xiii

List of Listings xiv

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Goal . . . 3
- 1.3 Scope of the Work . . . 3
- 1.4 Structure of the Thesis . . . 4

I Theory of Dynamic, Interactive Virtual Environments 7

2 Dynamic, Interactive Virtual Environments 9
- 2.1 Example Environments . . . 9
  - 2.1.1 A Therapy World . . . 10
  - 2.1.2 Crayoland - A Case Study . . . 11
- 2.2 Virtual Environments . . . 13
- 2.3 Context . . . 15
- 2.4 Dynamic, Interactive Virtual Environments . . . 17
- 2.5 Methodology . . . 19

3 Time 21
- 3.1 Philosophical Views of Time . . . 22
  - 3.1.1 Early Philosophy . . . 22
  - 3.1.2 Physics . . . 23
  - 3.1.3 Structure and Representation of Time in Computers . . . 23
- 3.2 Time in VEs . . . 26
- 3.3 Computer Animation . . . 27
- 3.4 Simulation . . . 28
- 3.5 Reactive Systems . . . 30
  - 3.5.1 Modelling Reactive Systems . . . 31
  - 3.5.2 Verifying Reactive Systems . . . 32
- 3.6 Modeling Checking and Temporal Logic . . . 32

3.6.1 Time extensions to Petri Nets . . . 33
3.6.2 Timed Automata . . . 35
3.7 Hybrid Systems . . . 35
3.8 Temporal Reasoning . . . 36

4 Dynamics **39**
4.1 Taxonomy of the Dynamics Design Space . . . 40
4.1.1 Dynamics Categories . . . 42
4.1.2 Implications . . . 49
4.2 Time-Based Categorization . . . 50
4.2.1 Categorization . . . 50
4.2.2 Implications . . . 54
4.3 Requirements for Dynamics Support . . . 54

5 Dynamics and Interaction **57**
5.1 Dynamic Interaction . . . 58
5.1.1 Dynamic Interaction in Classical VR . . . 58
5.1.2 Design Space of Dynamic Interaction . . . 60
5.1.3 A Time-based Taxonomy of Dynamic Interactions . . . 63
5.1.4 Implementing Dynamic Interactions . . . 64
5.1.5 Requirements . . . 67
5.2 Interactive Dynamics . . . 68
5.2.1 Illustrative Examples . . . 69
5.2.2 Design Space of Interactive Dynamics . . . 71
5.2.3 Implications . . . 74
5.2.4 Requirements . . . 75
5.3 Dynamic Interaction with Dynamics . . . 75
5.3.1 Illustrative Examples . . . 76
5.3.2 Implementing Dynamic Interaction with Dynamics . . . 78
5.3.3 Discussion . . . 82
5.3.4 Requirements . . . 84

II System Development and Implementation **85**

6 Virtual Reality and Existing DIVE Support **87**
6.1 Virtual Reality . . . 87
6.1.1 Basics . . . 88
6.1.2 Hardware . . . 94
6.1.3 Software . . . 102
6.2 Support in VR Systems . . . 104
6.2.1 Dataflow Systems . . . 104
6.2.2 Constraint Based Systems . . . 108
6.2.3 Simulations . . . 109

6.2.4 Entity, Actor, and Agent Based Systems . . . 109
6.2.5 High-level "Building Block" Systems . . . 110
6.2.6 State Machines . . . 110
6.3 Efforts to Model DIVEs . . . 111
6.3.1 Specification and Modelling Languages . . . 111
6.3.2 Temporal Modelling . . . 113
6.3.3 VR as a Hybrid System . . . 113

7 Functional Reactive VR 115
7.1 System Specification . . . 115
7.1.1 Use Cases . . . 116
7.1.2 System Requirements . . . 118
7.2 Design . . . 121
7.2.1 Selection of a Basic Approach for the System . . . 122
7.2.2 System Architecture . . . 124
7.2.3 Functional Reactive Virtual Reality . . . 130
7.3 FRVR Implementation . . . 132
7.3.1 FRVR VR Juggler Libraries . . . 132
7.3.2 FRVR Data Exchange Libraries . . . 133
7.3.3 FRVR FRP Libraries . . . 135
7.3.4 FRVR AVANGO libraries . . . 140
7.4 System Flow . . . 144
7.5 Developer Work Flow . . . 146
7.6 Discussion of Resultant System . . . 148
7.6.1 Dissertation Intent Derived Requirements . . . 148
7.6.2 Software Engineering Requirements . . . 149
7.6.3 DIVE Requirements . . . 151

8 FRVR Extended FRP 153
8.1 Requirements . . . 153
8.1.1 Improvements to Yampa Components . . . 154
8.1.2 FRVR Required Extensions . . . 156
8.1.3 Usability Requirements . . . 157
8.2 Improvements to Yampa Functionalities . . . 158
8.2.1 Simulation Kernel Time Resolution . . . 158
8.2.2 Sub-sampling Behaviors . . . 160
8.2.3 Sub-sampling Switch Transitions . . . 162
8.2.4 Time-aware Switching . . . 164
8.3 Extended Basis Functionalities . . . 165
8.3.1 Extensibility of the Yampa Modules . . . 165
8.3.2 Traditional Animation Tools Support . . . 167
8.3.3 Time Manipulation Functionalities . . . 168
8.3.4 Undo . . . 170
8.4 Transition Handling . . . 171

8.4.1 Conceptual Foundations . . . 171
8.4.2 Adjoin Transition . . . 177
8.4.3 Blending Transition . . . 178
8.4.4 Blending Adjoin Transition . . . 180
8.5 Reduction of Programming Difficulties . . . 181
8.5.1 Haskell Functionality Wrappers . . . 182
8.5.2 Visual Programming . . . 183

9 Examples **187**
9.1 A First Dynamic . . . 188
9.2 Dynamics and Pseudo Dynamics . . . 192
9.2.1 Simple Dynamics . . . 192
9.2.2 Pendulum . . . 194
9.2.3 Interactive Pendulum . . . 199
9.2.4 Newton's Cradle . . . 201
9.3 Follower - A Time Delay Behavior . . . 204
9.4 Boids . . . 206
9.5 A Full DIVE . . . 211
9.6 Discussion . . . 213

10 Conclusion **215**
10.1 Discussion . . . 215
10.2 Contributions . . . 216
10.2.1 The Dynamic, Interactive Virtual Environments Design Space . . . 216
10.2.2 Software Framework to Support Creation of DIVEs . . . 218
10.3 Future Work and Research Directions . . . 219
10.3.1 Validity of the DIVE Conjecture . . . 219
10.3.2 Research on the Confluence of Dynamics and Interaction . . . 220
10.3.3 Improvements to FRVR . . . 221

III Appendix and Bibliography **223**

A Functional Reactive Programming **225**
A.1 Concept . . . 225
A.2 Paradigm Context . . . 226
A.3 Haskell and Arrow Programming . . . 228
A.3.1 Haskell . . . 228
A.3.2 Arrows and Arrow Syntactic Sugar . . . 232
A.4 Yampa: Arrowized FRP . . . 234
A.4.1 Implementation Details . . . 234
A.4.2 Built-in Functionalities . . . 238
A.5 FRP in Use . . . 243

B Visual Programming **245**
B.1 Visually Programming Signal Functions 246
B.2 A Visual Syntax for Arrows 247
B.3 The Arrows Visual Programming Environment 249
B.3.1 Primitive Boxes 250
B.3.2 Compound Boxes 250
B.3.3 Networks of Ports 251
B.4 Implementation Details 252
B.4.1 Language of Implementation 252
B.4.2 Saving/Loading Visual Code 252
B.4.3 Haskell Code Export 253
B.5 Visual Programming of Arrows as Higher-order Objects 255
B.6 Discussion of the VPE 256

Bibliography **259**

List of Figures

2.1 A Scene from the VR Application Crayoland. 12
2.2 Example of Mindmap Style of Categorization Diagram 20

3.1 Allen's 13 Basic Relations in Interval Algebra 25
3.2 Hybrid Discrete Event + Continuous Simulation 30
3.3 Time Petri Net Example 34
3.4 Timed Automata Example 35

4.1 Taxonomy of Dynamics 41
4.2 Dynamic Scene Attributes Taxonomy 43
4.3 Taxonomy of Singular Object Dynamics 45
4.4 Dynamic Category: Propagating Quantities 47
4.5 Dynamic Category: System State 48
4.6 Dynamic Category: Abstract Components 49
4.7 Dynamics Taxonomy - Common VR Dynamics Highlighted 51
4.8 Categorization of Dynamics with respect to Time 53

5.1 Dynamic Interaction Taxonomy 61
5.2 Dynamic Interaction Time-based Taxonomy 65
5.3 Crayoland VE - the Butterfly Interactive Dynamic 70
5.4 Interactive Dynamics Taxonomy 73
5.5 Diagram of a Walk Cycle Animation 80
5.6 Potential Direct Manipulations of an Interactive Dynamic 82

6.1 Standard VR Interaction Loop 90
6.2 Bowman's Interaction Taxonomy 91
6.3 User Wearing a Head Mounted Display (HMD) 96
6.4 User in an Immersive Projection System 98
6.5 Typical VR Inputs Devices 101
6.6 Typical VR System Flow 103
6.7 Dataflow in the AVANGO VR System 105

7.1 Scene Graph Coupled FRP/VR Integration 126
7.2 Lock-step FRVR Implementation Diagram 128

7.3 Free-wheeling Threaded FRVR Implementation . . . 129
7.4 FRVR Implementation Diagram . . . 131
7.5 The Yampa Control Loop . . . 137
7.6 Information Flow in a FRVR-AVANGO Example . . . 141
7.7 FRVR-AVANGO Multi-field Reflection Class Diagram . . . 143
7.8 Dataflow in the Extended FRVR-AVANGO System . . . 144
7.9 FRVR-AVANGO Pairing Filed Connection Extension . . . 145
7.10 FRVR Timing and Data Flow Diagram . . . 146

8.1 Sampling Issues On Hand a Pendulum . . . 155
8.2 Sub-sampling Simulation Kernels . . . 161
8.3 Time Leaks Due to Low Resolution Sampling of Transitions . . . 162
8.4 Sub-sampling the Switching Event . . . 163
8.5 FRVR-Yampa Code Structure Diagram . . . 166
8.6 Keyframing Technique Illustration . . . 167
8.7 Two Example Behaviors, a and b. . . . 172
8.8 AFRP Switch Transition from a to b at Time $t = 3$. . . . 173
8.9 Possible Transition Timings . . . 175
8.10 Diagram of Building a Yampa Function . . . 183
8.11 Visual Programming Environment for FRVR . . . 184
8.12 Basics Components of the VPE . . . 185

9.1 VPE Programming of an Object Launcher . . . 193
9.2 Transitioning Pendulum Example: Sub-sampling Method . . . 196
9.3 Manipulate Pendulum Example Architecture . . . 201
9.4 User Interacting with a Newton's Cradle Simulation. . . . 202
9.5 Illustration of the Newton's Cradle Simulation Structure . . . 203
9.6 Delayed Replay of a Behavior . . . 205
9.7 Behavior Structure for a Flock of Boids . . . 207
9.8 Interactive Experience in a Middle Age Castle . . . 211

A.1 Haskell Functions as Processing Units . . . 229
A.2 Basic Arrow Functionalities . . . 232
A.3 Yampa Simulation Kernel: System Loop . . . 235
A.4 Yampa Reactivity: Switches . . . 241

B.1 A Network Formed from Primitive "Boxes" . . . 247
B.2 Basics Components of the VPE . . . 248
B.3 The Arrows Visual Programming Environment . . . 249
B.4 First and Second Arrows . . . 252
B.5 Example Network in the VPE . . . 254
B.6 Higher-order SFs in the VPE . . . 256

List of Listings

7.1 AxisAngle to Quaternion Conversion . . . 140
8.1 Time-Skewing Function Signatures . . . 169
8.2 Undo Function Definitions . . . 170
8.3 Adjoining Transition Behavior: switchWait1st . . . 178
8.4 Adjoining Transition Behavior: switchWait1stTimed . . . 178
8.5 Internal Event Blending Transition Behavior: switchTtimed . . . 179
8.6 External Event Blending Transition Behavior: kswitchTtimed . . . 179
8.7 External Event Blending Adjoin Transition Behavior . . . 181
9.1 VR Juggler/OSG Initialization of a Simple Box Dynamic . . . 188
9.2 VR Frame Loop Additions for a Simple Dynamic . . . 189
9.3 Haskell Simulation Setup Code . . . 190
9.4 Creation of a Behavior of a Box Moving at Constant Velocity . . . 190
9.5 Critical Components of Build System . . . 191
9.6 Simulation of a Launched Object. . . . 193
9.7 Code for Simulating a Pendulum . . . 195
9.8 Detection of Pendulum Velocity Transition . . . 197
9.9 Simulating a Pendulum Using a Transitioning Approach. . . . 198
9.10 preFrame Update Modified for User Input for Interactive Dynamics . . . 199
9.11 Simple Direct Manipulation in FRVR . . . 200
9.12 Switching States of the Direct Manip Pendulum . . . 201
9.13 Simulation of a Flock of Boids . . . 208
9.14 Steering Behavior Calculations . . . 209
9.15 Combing the Individual Steering Behaviors . . . 210
9.16 The Boid Data Type Definition . . . 210
A.1 Definition of a Simple Haskell Data Type. . . . 230
A.2 Haskell's Mathematical Style of Syntax . . . 231
A.3 An Arrow in Point-free Style. . . . 233
A.4 An Arrow in Point-wise Style (With Syntactic Sugar). . . . 233
A.5 Signal Function Type Definition . . . 237
B.1 Output Code for a Compound Box . . . 254

Chapter 1

Introduction

Virtual Environments enable the creation of fantastical worlds that come to life. When coupled with Virtual Reality technologies, those environments can be realistic enough for the viewer to believe they are a part of that world. This creates a powerful tool that can be used for a wide array of applications. As the environments are synthetically generated, the possibilities are nearly endless in what can be created. Those possibilities are limited only by the imagination of the designer, the power of the computers, and the sophistication of the programming that enables it to come to life.

The medium of Virtual Environments has become common place, from movies to computer games. Virtual Reality (VR) technology appears to be coming of age. Improvements to the technologies of VR are widening their applicability, and the cost of the technologies is reaching a point where it is affordable for many applications, including home usage. Computer games perennially show that the field has little lack of imagination. The one area that is still lacking, even for specialists, is the programming structures that enable the modelled environment to come to life.

That the programming capabilities have not kept pace with 3D modelling and rendering and hardware technologies is in many senses astounding. However, the reasons for this are multi-faceted. One of the most pervasive reasons is that VR researchers have focused instead on the plethora of additional factors, such as interaction, hardware, immersion, and presence. Other communities that work with Virtual Environments (VEs) have focused on the modelling and rendering. Among them is the gaming community. The gaming community has created wondrous VEs, including interaction and action, but the solutions they have produced are highly specialized to the needs of individual games (or genres). Few generally applicable solutions for supporting the creation of such vibrant worlds have been pursued.

This dissertation investigates the essence of Virtual Environments that "come to life" and the development of a solution that supports their creation. The work is based on the conjecture that such environments have to contain elements that change over time. Fundamental support for the creation of such dynamic content is lacking. However, the support of dynamics in isolation is not enough. Interaction with all

aspects must be accounted for. Interaction is not only one of the cornerstones elements of Virtual Reality, but the inclusion of dynamics seems to afford interaction. The successful creation of such environments must consider both time dependent phenomena and interaction. Therefore, the environments of focus in this dissertation are referred to as "**Dynamic, Interactive Virtual Environments**."

1.1 Motivation

The need for more interesting and interactive Virtual Environments has been formally noted by various people [CEH03, CKP95, Del00, DR03, ESYAE94, Gei98, Gre96, Mur97, PST+96, SK03, Whi03, Wil00, WGW90, Zac96]. The areas which already see the need for these kinds of advanced environments are varied. The technical fields of engineering and the sciences have been striving to use virtual representations of their data spaces to gain insight into their problems for years. One of the areas of great interest and potential is the exploration of problems that are time dependent, but also one that has had limited success. Two of the largest remaining difficulties are interaction and dynamics. The natural sciences (particularly Psychology, Sociology, and Cultural Heritage) are starting to investigate the usage of VEs. Here the reproduction of specific situations and places is desired. The area of entertainment, especially computer games, is also highly dependent on the creation of compelling environments. Across all of these fields, increasing the interactivity and dynamicity of VEs will bring benefits. The utility of VE based technologies is likely to expand as such environments become more commonplace, meaning adoption in larger realms is likely.

Supporting software mechanisms for the creation of VEs in both VR and in related areas are not new. Numerous attempts have been made to provide better support. However, the focus of those developments has predominately been on other aspects, such as dealing with VR hardware or maximizing realism in rendered graphics. A few select efforts have set out to improve environment content. The current model of creation of Dynamic, Interactive Virtual Environments (DIVEs) involves very specialized programming. The standard method is direct programming of dynamics and interaction in low level programming languages, typically C++. Even these experts have difficulties creating environments can be deemed DIVEs. As a result, the environments presented are typically limited to largely static structures with special objects of interest added depending on the expertise of the programmer. Commonly, this is limited to "point and click" style interaction and direct manipulation in VR environments.

Some solutions have been proposed that focus on environments of interest here. They have often be specific to a special problem or been broad in goals. Those solutions sought to create "the Metaverse." For many, the Metaverse is the grail of VEs. It is a fictional world from Neal Stephenson's book, *Snowcrash*. The Metaverse is a completely immersive interactive world of a compelling nature. The various existing developments have either ended up too narrow in scope or failed to provide the desired effects due to the breadth of support attempted.

The software support issues that exist are compounded greatly by the expanding community that desires to create and take advantage of such DIVEs. These people come from many various backgrounds, with little if any programming experience. The authors' personal experience, even with engineering graduate students with a programming background, showed that the creation of anything more than a static environment was overwhelming. The methods that sufficed for programming experts from the field were hard to grasp and were only learned at great expense. In those experiences, and having to program in those ways, the desire for better support was born. Some of the difficulties may be inherit in the problem space, but support should be possible if properly addressed.

1.2 Goal

There are two goals to this dissertation: gaining an understanding of the design space of Dynamic, Interactive Virtual Environments and the identification and development of appropriate supporting software for the development of those environments. To gain insight into what DIVEs actually are and what they may entail, an investigation delivers an initial understanding of the field and its potentials. A definition of the kind of Virtual Environment that is desired is developed. Based on that, an exploration of the major design spaces that compose DIVEs is undertaken. The result of this exploration of the space is a very large and diverse set of possibilities. Categorizations of the possibilities increases basic understanding of the possible content of such environments and also provides insight into the support needed for their creation. Finally, requirements for support based on the complete investigation are developed.

The second goal of the work is to develop better support for the creation of the environments specified in the initial investigation. This support should not be exclusively for the modelling of such an environment, but should provide support in the implementation of such environments. The system should improve on existing systems by being better matched to the actual needs of people creating such environments. This entails both providing direct programming support for the functionalities that are required and by providing a programming model that matches the creators understanding of the content they are creating.

1.3 Scope of the Work

The work of this dissertation contains portions that cover a very broad space and the endeavors undertaken are large in scope. In order to insure the dissertation was possible to complete, the scope of the dissertation was constrained in some aspects. This section describes the scope of the work that was undertaken.

The definition and exploration of the design space DIVEs was restricted to probing of the experience of the group at the university. Little relevant published work exists.

Since this work is the first to take a formal approach to defining the design space, the analysis is in some places is incomplete. Even the design spaces considered are restricted to the combination of dynamics and interactivity. Application directions that extend this space even further, such as Interactive Storytelling, are not considered in depth. The investigation of the design space turns up a number of avenues that have remained unconsidered to date. Because these fields open up into large, newly defined spaces, the research and developments are mostly restricted to avenues that can be fully defined at this point.

Based on the understanding of the support needs for creating such environments that is developed out of the study above, a support system is developed. In this development the scope of the work had to also be constrained in a number of ways. To find an appropriate method of supporting DIVE creation, a survey of time in computer science was performed. This included a broad search across many fields for methods of potentially supporting DIVE creation. In this work, the major applicable directions found are presented. Areas that were researched, like cognitive psychology, but where no directly relevant information to implementation support was found, are not presented here. A broad survey of existing approaches taken in VR and related fields contribute to the understanding of the field and a necessary analysis of the design provides a frame for achieving a reasonable system. Because so many works could be considered on some level to be relevant, a complete review is not performed. A survey of the basic methodologies that have been taken is instead provided.

A system to support the identified requirements for Dynamic, Interactive Virtual Environments is developed. The support of all possibilities is beyond the scope of this work. The largest compromise on this front is the level of support. Instead of attempting support for every user, the support is restricted to those with a significant mathematical background. This compromise ensures that the support is available to a larger community, without striving for currently unattainable generality and ease. This restricts the developments to a kind of middle-ware, while the requirements of increased ease make existing VR approaches unusable.

Finally, the examples illustrate the power and complexities possible with the developed system are limited in scope. Stephenson's Metaverse is not presented, but the examples highlight the basics of how such a world would be built. The examples demonstrate how the system covers the requirements laid out in the analysis of the Dynamic Interactive Virtual Environments design space. While not every possibility can conceivable be covered, examples highlight the major classes of effects that are possible in the environment.

1.4 Structure of the Thesis

The thesis is divided into three parts to provide further structure its ten chapters and two appendices. Part I defines the type of environment of interest, Dynamic, Interactive Virtual Environments. The design space of DIVEs is explored there. Part II develops

a system to support the creation of DIVEs. Part III contains the supporting materials in the appendices and the bibliography.

Part I explores the design space of the systems that are the center of our investigation. It is divided into four chapters. The first chapter introduces and specifies Dynamic, Interactive Virtual Environments. The design space of what they potentially consist of is developed at a high level. Chapter 3 presents critical background to the development of these environments, an understanding of time. The meaning and representation of time, with a particular focus on how it is viewed in Computer Science, is presented there. Additionally, the chapter presents a survey of the handling of time and implementation of time dependent components in various sub-fields of Computer Science. Chapter 4 defines and analyzes the design space of Dynamics, a core component of this dissertation. Chapter 5 introduces three new concepts, *Interactive Dynamics*, *Dynamic Interaction*, and *Dynamic Interaction with Dynamics*. Both chapters analyze the area and categorize the design space into taxonomies. These taxonomies provide structure to consideration of the areas and insight into their nature. A requirements analysis for systems to support building such environments is also developed based on the taxonomies and prior analysis in the respective chapters.

Part II presents the implementation of a system to support the creation of Dynamic, Interactive Virtual Environments. Chapter 6 begins by clarifying the Virtual Reality context of this work and explains the relevant portions of the field for this work. The remainder of the chapter is dedicated to a survey of the handling of time dependent and interactive environments in existing VR systems. The design and implementation of a system of support is presented in Chapter 7. The resulting implementation is referred to as Functional Reactive Virtual Reality (FRVR). Chapter 8 considers extensions to the Functional Reactive Programming system that is the basis of FRVR that make it more powerful and easier to use. Chapter 9 demonstrates the usability and power of the FRVR system with extensions through a number of selected examples. Chapter 10 concludes the dissertation with a discussion of the work, a listing of the contributions to the scientific community, and a look at research directions and future work identified.

Part III contains additional matter that supports the main work. Appendix A presents the Functional Reactive Programming paradigm and details of the Yampa implementation. It also provides a cursory introduction to the Haskell language and the Arrow extension to Haskell. Appendix B presents work done on a visual programming environment for Arrowized Haskell, including FRVR code. It was developed to make programming of FRVR more accessible to a wider community. Finally, the complete bibliography is included.

Part I

A Theory of Dynamic, Interactive Virtual Environments

Chapter 2

Dynamic, Interactive Virtual Environments

The creation of Virtual Environments that go beyond a static place of exploration is the central theme of this dissertation. The creation of environments that capture the interest of the user for many hours is possible, as witnessed by modern computer games. The goal is to make the creation of such interesting, experiential environments easier to create. However, a fundamental understanding of what kind of environments we want to create has neither been defined nor analyzed.

This chapter is the first of a series of chapters that investigate what makes up such environments. In this chapter an understanding of the type of environment of interest is established. After examples that are illustrative of the type of environment in consideration are presented, a definition of the environments that are being addressed will be advanced. Components of such environments that make them interesting and exciting are identified. These components are further investigated in the following chapters.

2.1 Example Environments

To better understand the kind of environments that are of interest, this section presents two examples. Both of these ideas are presented via short discourses. The main mechanism will in the form of short stories that highlight the use and environment together. The examples are, by necessity, short and cover only a small fraction of the possibilities. They are also examples of persuasive applications that are *affective*, and in many senses part of the "story telling" class of applications. The importance of them is not the elegance of the content, but rather what characterizes the content from a system side and makes it possible for them to be so affective and effective. The contexts in which such environments may be found is further explored in the next section.

2.1.1 A Therapy World

"Sir, are you ready for your session today? I will turn on the device in a moment. When I do, you will be immersed into another world. Please take a moment to become comfortable with the environment that you will be entering today and with the interface. In a bit, we will start the session."

> She came in from the door ahead. "We didn't know what to make of this, so we called you in. Please go in and see what you can make of it all." With that she disappears, looking a bit pale, out the only other door in the room, behind you. Only one thing to do, check it out.
>
> You go to the door and open it. The hinges creak slightly as you enter. Not uncommon in these old buildings I guess. Guess I'm just jumpy because of the woman's demeanor. The narrow and dank looking hallway goes forward. The droplets of water falling sparkle in the light of torches. Torches? Burning on the wall and seem to be my only lighting.
>
> Rounding the corner it becomes obvious what disturbed the women earlier? The room here is full of motion. There is no one in sight though and how on earth did this all get built by them, back then? The dancing shadows cast from all the motion are dancing but don't show the other side of the hall. The light from the single torch on the way next to you is not enough.
>
> Grabbing the torch from the wall you proceed to investigate. A ball bouncing up and down, as if by magic...as you approach the hiss of air gives a clue to how that is happening. Somehow everything you see is like child toys, albeit old and kicking up a lot of dust. There is something unwholesome about the situation. Out of the corner of your eye you catch sight of something different. Was that a passageway that is now covered up? You watch and note that the wall there seems to be moving. It's hard to tell though with the shimmer of mist between here and there that seems to grow and ebb. You'd also swear that something keeps appearing and disappearing where you thought you saw the passageway. It seems a boulder solidifies directly before the passageway slowly, then quickly disappears. The cycle seems to take a minute, though you aren't sure it is always the same.
>
> Just have to time it to get through before the boulder starts coming back again...1...2...

"Ouch. That hurt"

"Is that better?"

"Yes, blast! I missed the chance to get by that thing!"

"Settled down sir, we can set it back a bit in time, say one minute."

"What? oh right..not real. Can we set it back two minutes?"

> The shimmer of mist grows up in front of you. Oh ja, this was before I had approached that strange boulder...

The above little story is a description of an emerging use of Virtual Reality technologies. The story could be part of a therapy session, where the user experiences mild discomfort up to pain. Such an event is not yet really reality, but is a possibility that isn't too far off. Actually, nothing technical really stands in the way and research showing that VR as an "anesthetic" can work exists [GKKR05]. However, for it to work, the user has to be immersed in another world (Chapter 6 discusses the technologies of VR that make this possible). Action and interaction make the world described above into something that captures the user's attention at a level that they mentally find themselves absorbed into that world instead of in a painful therapy session.

2.1.2 Crayoland - A Case Study

The display is dark, fitting of a device called a CAVE™. With the start of the application, the vivid colors of a field surrounded by craggy mountains pops into existence. The initial impression of the display surrounding you is very impressive. You cannot help but look around, both the take in the environment surrounding, but also just a need to grasp where you find yourself. After getting a bit adjusted, the guide starts to describe technologies involved as the world starts to move under you via the guide's control. You quietly nod, not knowing if she thinks you are really listening. You definitely aren't at this point, getting adjusted to the feelings caused by the world moving under you. "Check out the house" comes a command. "just move around and view it. Stick your head through the window and look around." Hesitantly you find that you can do it, lean in through the window.

Cool, but what now? Across the field the guide brings you to a pond. There is a frog croaking. Next he brings you close to the tree. Hanging from the tree is an innocent looking beehive. "Take the controller and hit the beehive." Uncertainty wells up...but it is just imaginary right? OK, the virtual hand attached to the controller swings with the controller and makes contact. Suddenly the buzzing of bees is all around! Bees angrily fly around. You step back; they follow. After a while of high speed moving by the guide, safety is reach, the bees have quit following. You note your heart still beating quickly.

Over in another part of the field there are a flowers growing. Definitely more calming than the nasty bees (...though I guess I cannot blame them, after all I did hit their hive). Ah, a butterfly flying around. "Take the controller again. Hold out your hand near the butterfly and keep it still." After the previous time, skepticism abounds, but alright you do it. After a few seconds the butterfly lands on the back of your hand. Did you just feel it land? It stays there for a while, occasionally adjusting its wings. You move a bit too much, and the butterfly takes off.

The world described above is a land called "Crayoland" and it is the first application written for the CAVE™ VR systems(see Chapter 6) as a demo [Pap98, Pap]. It is the

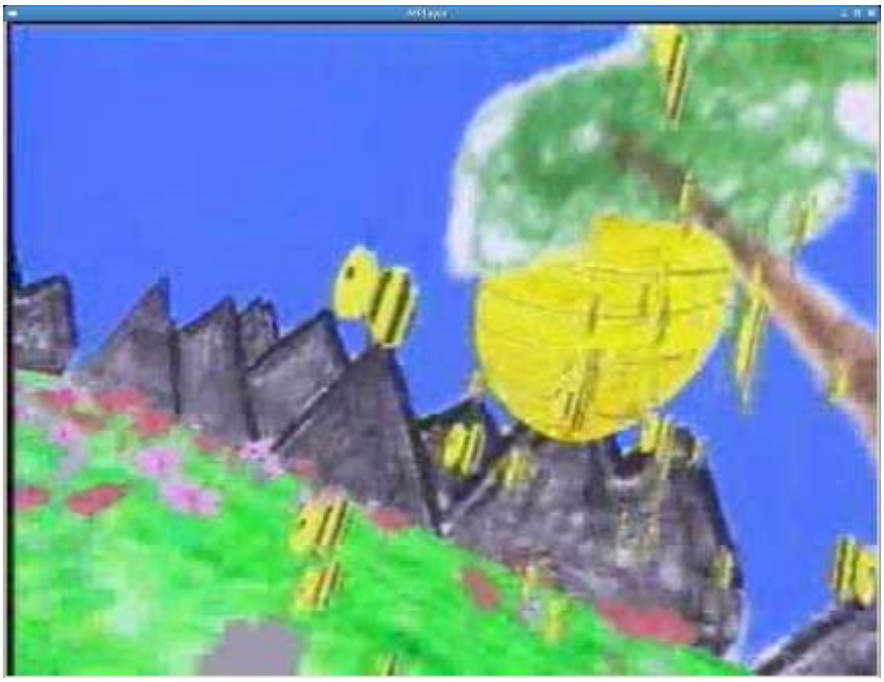


Figure 2.1: A Scene from the VR Application Crayoland.

first immersive application many people see, even today. The description above is roughly the authors experience as he first experienced VR and such an environment. The author is, however, not alone in finding it to be quite an experience; many VR experts still find it to be the best application. Ironically, it is also extremely unrealistic graphically. This can be seen in Figures 2.1 and 5.3.

What makes Crayoland so interesting? The experience. Even for those who don't see it as their first immersive application, it is unequivocally a memorable environment. The reason definitely is not just the charm of the crayon drawing. They contribute to the experience, as much as the choice of content, like the butterfly. However, the world only functions because the world reacts to the user and is full of life itself. The first person perspective of the house is interesting, but when Crayoland becomes reality is during the batting of the beehive and the ensuing pursuit of angry bees (are they really angry? It is a safe bet to say that every visitor to Crayoland says they were.) or getting the butterfly to land on your hand (it actually lands on a virtual hand representation that is in front of the control device, as seen in Figure 5.3, but people react as in the description above, saying that it lands on their hand). Without the parts of the world that are in action and react to the user, Crayoland would be long forgotten.

In the context of this dissertation, Crayoland exemplifies a possibility that has always been there, but is frustratingly rarely present. Crayoland has a few sounds and what is really fairly simply created behaviors and interactions, but remains a rarity. Often the focus of more current demos (when they are developed) is humanoid char-

acters or great graphics. In the ten years, since the author first saw Crayoland, he has searched for more compelling environments and seen many interesting environments. However, if any one demo should be picked as the best, Crayoland wins hands down. Why don't we have more Crayolands and better ones?

2.2 Virtual Environments

The previous section has illustrated the kinds of Virtual Environments of interest in this dissertation. In this section, a more formal look at Virtual Environments in general is provided as a foundation for this work. Virtual Environments are a foundational component for a number of application and research areas. Although commonalities exist across all areas, what a Virtual Environment is and how it is defined differs between those communities. The term Virtual Environment was even used for a time as a replacement for the term Virtual Reality (a specific way of presenting a VE that is described in depth in Chapter 6). A broadly applicable definition is presented here and will be the foundation of this work. Since an understanding of Virtual Environments is typically left implicit, a short discourse into what Virtual Environments is undertaken. Further examples from the areas of interest in this work can be found in Sections 5.2.1 and 5.3.1, as well as the examples produced in this work in Chapter 9.

We define a Virtual Environment as:

> A Virtual Environment is a synthetic, computer generated, spatial (3D) environment, which may or may not reflect the "real world."

This definition is generally applicable across all areas that rely on such environments. The definition, though, differs from the standard definitions; for example Bowman et al. describe them as "a synthetic, spatial (usually 3D) world seen from a first-person point of view" [BKLP05]. VE is sometimes synonymous with 3D environment; the environment being 3 dimensional in nature is almost always a component of the understanding of VEs (2D game environments may be the arguable exception). An important (for some critical) descriptor is purposefully absent from our definition: visual. Our definition purposefully eliminates specifying a modality of presentation. Although such environments are today almost exclusively visual environments, such a limitation is artificial and the nature of the environments is not explicitly dependent on the modality of presentation. For instance, a number of VEs that are only presented via sound have been created.

The idea that the VE is either synthetic or computer generated is also commonly present, but not is combination. Used together, they exclude a number of unrelated ideas, such as dioramas, that would otherwise fit the definition. Finally, the addition of the reflection of the real world qualifier is added in this work to emphasize this fact. We feel this is important as there is an assumption in many realms that only "photo-realistic" worlds that strive to recreate the physical world's properties are valid. This view is very narrow and many of the contexts of interest this is not a valid assumption.

Virtual Environments are found in a number of contexts. Probably the most well know and wide spread usage of them is in computer games. Virtual Environments in games are varied in nature. The VEs of games often exceed those of other areas in terms of their properties (discussed below). Virtual Reality is another context often seen as related to games and the focus of this work. The number of different areas of application of Virtual Reality means that VEs of very different characteristics are often created there (see Section 2.3 or any of various VR books [BC03, BKLP05, Rhe91, SC03]). Another well known area that uses VEs is the film industry. Although termed "CG" films, these are VEs that are presented in a non-interactive way. Similarly the mixed reality of CG special effects creates hybrid environments that are often completely virtual.

It is somewhat surprising, given how widely spread the usage of VEs is, that there seems to be little work on defining the characteristics of them. Since a focus of this dissertation is on defining a special set of VEs, a small collection of characteristics of VEs is collected. A thorough investigation is outside of the realm of the work of this dissertation, so this should only be seen as a framework for discussion. We define the basic characteristics of Virtual Environments to be:

- dimensionality (typically 3D but could be higher)
- extent
- scale
- modalities represented
- modalities presented
- fidelity (across each modality)
- reality spectrum - real $\leftrightarrow$ complete synthetic (abstract)
- presentation style
- interactivity allowed
- dynamicity present

These characteristics parameterize the spectrum of VEs. The dimensionality of the VE and the extent (size) of the VE are fairly obvious characteristics. The scale of a VE is more semantic in nature, describing whether the VE scaled differently than it is experienced; an example of this is the visualization of nano-tech at the size of meters. VEs may be defined in more than the visual modality as explained above. Although not restricted to them, usually one considers the human senses when speaking of modalities. Related to this is the modality of presentation (which may not always be the same as the modality of the VE, e.g. sonification). The fidelity of the VE deals with how precise and accurate the VE is represented. It is important to note that the fidelity of the environment is independent of the "reality" of the environment. The presentation style of the environment is how the world is displayed to the user. This is analogous with placing filters in front of a camera to change the appearance of the environment. Finally, two elements of VEs that are of particular importance in this work are interactivity and dynamics. These will be discussed extensively in the following sections and chapters.

2.3 Context

Virtual Environments have been developed for and used in many different application areas. Almost all of these areas have a vested interest in environments that go beyond static worlds. Many of them hinge on such environments. In this section an overview of the applications areas that are reliant on such environments is presented. The focus will largely be on real-time interactive applications, in particular those that presented via Virtual Reality technologies. Reviews of application areas can be found in most VR books [BC03, BKLP05, Rhe91, SC03], but consideration of aspects those applications that are of interest for this dissertation are not explicitly covered.

The application areas identified that utilize environments that exemplify those that we are concerned with are:

- Entertainment
- Edutainment
- Training
- Clinical Psychology
- Virtual Reality Exposure Therapy (VRET)
- Sociology
- Physical Therapy
- Digital Heritage/Cultural Heritage
- Architecture
- Visualization
  in areas like Engineering, Chemistry, Biology, Physics, Mathematics

Each of these application areas is discussed in the following paragraphs. A general idea of what kinds of environments they use for their goals is present. When not obvious, short examples of how the area tries to leverage advanced environments are provided.

The areas of entertainment and edutainment (sometimes referred to as Serious Games in current literature) both center on creating worlds that capture the interest of the user, though the goal of those environment differs. A concept that is often used in relation to the more advanced environments in these realms is that of an *interactive experience*. A key to creating interesting environments is moving beyond static environments and involving the user. Creating involvement is typically done by including interaction capabilities. Most often this reduces to a point and click kind of interaction. Environments are typically equipped with a small set of independently moving objects. The topics of computer games and edutainment are large; starting points for research in these areas include: [Bye07, Don07, EN06].

The area of physical therapy is one of the most promising emerging applications of VR. Many physical therapy treatments can be painful and enduring for longer periods of time. The goal of VR usage is to distract the user from the pain involved in the therapies. The typical setting of physical therapy applications is that of gaming environments, as they are among the few available environments that can successfully distract

the patient for longer periods of times. To be effective the environment has to be an engaging environment. Additionally, therapies can take time and generally involve multiple sessions. This means the environment(s) have to be engaging over longer periods of time. Overviews for this young area can be found in: [GKKR05, Riz06, WBSD+07].

Training uses of VR are now various. Training is related to edutainment, but focuses on having a high transfer of learned skills. Training is one of the earliest usages of VEs, i.e. flight simulators. The military is embracing virtual training as a way of training in places not reachable and situations that are too dangerous to train in reality. Likewise, some companies are developing training for dangerous jobs or those where errors cost too much. The medical community is also adopting related technologies for the training of different treatments and operations.

Insightful training examples for our context are driving simulators. The simulation of the vehicle and its travel through the environment is typically only a small portion of such training systems. Simulations are designed to test aspects such as visibility, driver distraction, etc. Traffic simulations, both emergent and also strictly timed events are required for the testing. For instance, a study may desire to test how the driver will react when another car is approaching a crossing at the same time, requiring the simulation to make the event occur at the proper time. This requires knowing when to start another car, and running it to the corner over time. Here we see how timing of events in such environment is critical to their effectiveness. Introductions to the traffic simulation community can be found in [BFP96, CKP95, Wil00].

As we saw above, simulations in training are used to provide controlled environments. In Psychology, VEs are becoming a viable method for providing controlled environments for their experiments. This has large advantages of having the immersive nature of VR and higher presence. As with the simulations above, much of what the Psychologist wants to control is the timing of events or length of exposure to certain stimuli. VR provides the possibility to do this. A good reference for this area is "Section II: Virtual Reality in Clinical Psychology" of [RWM98].

Beyond usage in Clinical Psychology, there is a movement to use VEs and specifically immersive VR technology for the treatment of phobias. Part of its power is that the Psychologist is in control the environment in a way that is not easily possible in the real world. That the user is at some level aware that the environment is not real is also exploited to provide "in vivo" exposure therapy. This puts them at ease, at least to some extent. Interestingly, in the area of Virtual Reality Exposure Therapy (VRET), there is evidence that VEs are more successful than classical in vivo therapies [KEOB04, PE08]. Even inter-relational aspects can be tested here with new control. This then involves having computer controlled avatars playing roles. An example of this can be seen in Pan's study [Pan07].

Applications of VR in the sciences and engineering often take the form of visualizations of problems and data. The field is quite established, with most techniques falling under the umbrella of "Scientific Visualization" [JH04, SML98]. Static visualization of data is well established in certain areas, for instance Geology and Fluid Dynamics.

However, for many problems incorporating the time dependent nature of the problem in the visualization provides benefits. Currently this is done by pre-calculation of each "time step" and then simply visualizing the results.

Along with real-time generation of time-based phenomena, interaction forms a highly desired and emerging possibility in visualization. In many cases, the scientist or engineer would like to make changes and see the effects of these in the generated VE. Most commonly, parameters of an underlying simulation that is being visualized are controlled by the user. This could be something as "simple" as changing a mass or changing the density of the material being simulated. Other usages include traditional spatial manipulation. Take the case of an engineer visualizing the flow of air in a kitchen. Ideally, they would like to arrange the air outlets in the environment to see the effect on the flow - which is being visualized. The visualization of such problems are also often dynamic (or at least based on it), meaning a combination of dynamics and interaction are required.

Architecture is an area that has experimented with use of VEs and is seeing some success [Why03]. Traditionally, architectural "walk-throughs" have been performed in buildings. This translates to having a model that is life size when viewed in VR. However, environments that are interactive are desirable, where changes to the structure itself or, at least, the spaces within can be manipulated. The earliest of these applications were for the design of office environments. Today, 2D systems for the layout of a room are available at many outlet stores. Another emerging use is "populating" the environments with virtual characters to provide a feel of "life" in the visualization. Related to this are efforts to perform urban planning with populated VEs.

Digital Heritage and Cultural Heritage form a broad area of VE applications, sometimes denoted as Virtual Heritage [CS04, JH07, Zor03]. In general this is an area meant to present historical places, artifacts, or times in digital format. Digital Heritage is concerned with time, i.e. to present things from a time past. Due to the historical significance of the subject of many projects, a static model is often intriguing, e.g. Notre Dame as it was in the 1600s. Most of the time, providing the place with a sense "life" is advantageous. In the case of Cultural Heritage, it is usually of critic importance. In most cases, the relationship that we, as people, have to these artifacts is drawn through the people of bygone cultures and their usage of the buildings and tools. Populating the environment with avatars is the most obvious way to achieve this, but also things like smoke filling a hall as it would have from a fire help set the context.

2.4 Dynamic, Interactive Virtual Environments

In the previous sections, examples of the type of worlds and discussion of the context in which such environments are needed highlight various things that could be used to make them interesting and exciting. In this section, the type of VEs that this dissertation focuses on is clarified and formalized. The components out of which they are composed

are identified. Those components will then be investigated in depth in the following chapters using a method described in Section 2.5.

Previous Virtual Environment was defined in broad terms. In this work only a subset of the VEs possible are of interest. The simplest VEs to create and control by computer are static environments. These environments are presented by nothing changes, neither from itself out nor by outside impulse. At the essence of all of the different VEs presented previously is the extension of the environment beyond this static nature in some way. Through such mechanisms the world presented is more interesting. Virtual Environments can be expanded on in one of two major ways to make them livelier: adding components to the world that change over time or adding the capability of interaction with the world interaction. Both of these ways are conceptually straight forward and both are more complex in the programming than it is often expected.

Adding components that change over time changes the static environment in ways that are potentially make it more interesting. The components that change over time will be called *Dynamics* in this dissertation. People are naturally drawn to moving objects. Programming time based changes to the environment turns out to be challenging. The use of computer for time based calculations is by no means new, but the requirements for Virtual Reality and related systems increase the difficulty. The biggest factor is that such systems are by necessity real-time systems and with variable update rates. In the next chapter, the topic of time will be covered, including an investigation of methods for handling time. In Chapter 4, Dynamics will be investigated in depth.

The ability to interact with the environment is a vital part of Virtual Reality and at the core of games. Interaction is one of the decisive differences between such areas as VR/games and film. Particularly within the Virtual Reality community, the most common way of making a VE more interesting is by allowing the user to interact in some way. The most basic interaction, and nearly omni-present, is the ability to move within the world. Unfortunately, the sole interaction of many VR based VEs is the ability to move through the world in a first person manner. Although, the power of this ability should not be minimized, the user is only spellbound by this for so long. Interaction with the environment has been a topic of VR for as long as it has existed. Interaction is also the one area that has had formal investigation. Generally, Bowman's work [BH99] is accepted within the community. Interaction in the context of this dissertation is taken to be a bit broader, the ability for the user to influence the environment. Since this area is already formally analyzed within the VR world, it will only be presented briefly in the context of Virtual Reality in Section 6.1.1.

The inclusion of dynamics and interaction in the virtual environments lies at the core of this work. In fact, we will define the environments of interest to be *Dynamic, Interactive Virtual Environments*. In other worlds we are interesting in Virtual Environments that contain both Dynamic components and Interactive components. An important question that needs to be addressed is what happens when components are both interactive and dynamic? This question is not only valid, but crucial to the development of better support of new interesting environments. Unfortunately, where

interaction and even to some extent dynamics have been formally investigated, the area of their combination is undeveloped. On considering the intersection of dynamics and interaction, two major directions are identifiable. One direction is interaction with dynamics, which we will term *Interactive Dynamics*. The second is interactions which induce a dynamic. These will be referred to as *Dynamic Interactions*.

The topics of Dynamic Interaction and Interactive Dynamics will be addressed in depth in Chapter 5. However, it is useful to first explain them roughly and set them in context. The realm of interactive dynamics includes a multitude of interactions that are intrinsic to Virtual Reality, though is not defined by them. These are interactions that occur over time from their nature, thereby inducing a dynamic in the environment or system. Basic components of VR such as head tracking and classical direct manipulation in VR are prime examples. The user's head movement is continuous as well as their hand movement. These interactions are well known in VR and related fields. Interestingly, the time component of them is only truly considered in one work, that of Steed [Ste06]. In that work he attempts to build a time based model of the selection interaction.

Interactive Dynamics can likewise be found in current VR systems, though typically of limited use. These are any time that the user interacts with a dynamic component of the world. The typical interactive dynamics are performed by selection of a dynamic object and an event (button press in VR) that modifies some aspect of the dynamic (often stopping or pausing it). Interaction with dynamics is an implicitly acknowledged concept within the VR and related communities, but is almost completely ignored in research. Interactive Dynamics are investigated in Section 5.2 of Chapter 5.

2.5 Methodology

In the following chapters the characteristics of Dynamic, Interactive Virtual Environments are to be defined and explored. The specialized Dynamic, Interactive Virtual Environments that are of interest are characterized by four types of content: Dynamics, Interaction, Dynamic Interaction, and Interactive Dynamics. The methodology used in the development of an understanding of these characteristic contents is explained in this section.

In order to gain an understanding of the design spaces of the different content types, the following methodological approach was taken. The initial step taken was to collect all the possibilities for each of the different content types. This included drawing on knowledge of past VR works, past and present computer games, a bit of creative thinking. The initial collection was developed by a group of VR experts with over 25 years of experience collectively. The collections demonstrate the multitude of possible content. Particularly in the case of dynamics, the possibilities are overwhelmingly abundant. In order to be able to deal with them all, categorizations of the possible content is necessary. Additionally, the content listed is limit to the concepts that

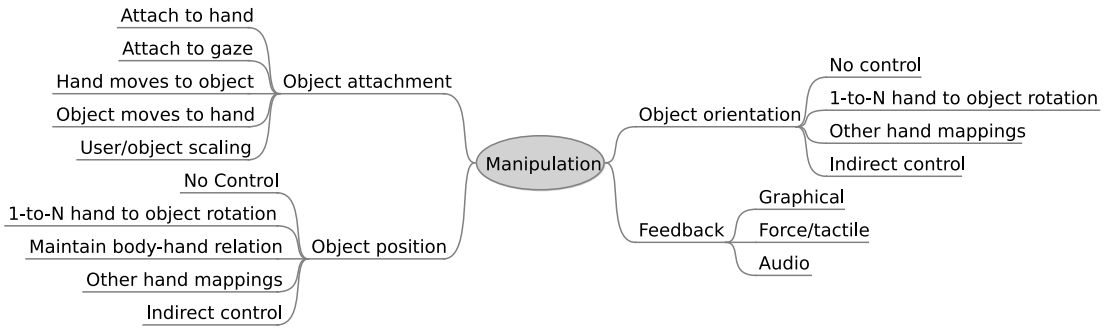


Figure 2.2: An example of a mindmap style presentation of a categorization based on Bowman's interaction taxonomy [BH99].

capture the essence of the content type instead of specific examples. Exceptions are taken in places where the examples are necessary to clarify an idea.

Various methods of building categorizations exist. The method of categorization chosen to follow was that building taxonomies of the design spaces. Taxonomies are already familiar to the VR community. Bowman's interaction taxonomy [BH99] is heavily cited and commonly accepted. Taxonomies are also an accepted categorization form reaching from ancient times. Taxonomies are naturally dependent on the approach that is used for how to classify the design space. This will also play a part in our taxonomies and using different views is informative. The method used to visualize a taxonomy can be varied. Bowman used a traditional branching structure, with sub-trees for his interaction taxonomy (discussed in Section 6.1.1). In this work "mindmaps" are used to display the taxonomies developed. An example of a mindmap taxonomy based on part of Bowman's taxonomy can be seen in Figure 2.2. The mindmap concept is explained in works like [Buz91].

Finally, after structuring the design space though the taxonomies, an analysis of the design space can be performed. The goal of this analysis is to generate a set of requirements for supporting the development of DIVEs. These requirements will be part of the input to the design of a system in Chapter 7 of Part II. In addition to the development of requirements, use cases that exemplify the areas will be identified both to inform the system development and also to provide specific test cases.

In the following chapter, an investigation of time in computer sciences is provided. The chapter covers both the aspects of time as they are relevant to the modelling and implementation of systems and also how time based aspects are handled in various fields of computer science already. In Chapter 4, the possible Dynamics of a VE are investigated, following the plan laid out here. The work of Bowman will be assumed to be enough to cover pure interaction. Section 6.1.1 presents that taxonomy, and details can be found in any of [BH99, BKLP05, Bow99]. The intersection of Dynamics and Interaction is handled in Chapter 5.

Chapter 3

Time

One of the core aspects to achieving the kind of Virtual Environments of interest identified in the previous chapter was Dynamics, changes to the environment that occur over time. The second core aspect is interaction, again creating changes that occur at some point in time or even over time. These two important processes are investigated in Chapters 4 and 5. The essence of these relies heavily on understanding the nature of the independent factor, time. The implementation of a system of support for Dynamic, Interactive Virtual Environments (DIVEs) which follows in the second half of this work, is also naturally highly dependent on understanding time. This chapter explores time's nature, particularly in a computer science perspective necessary for implementation. An overview of how different areas of Computer Science deal with time is provided as the basis for eventual decisions on how to support DIVE creation.

Time is a pervasive part of our lives. In the modern world, we are quite obsessed with time and are taught from a young age to develop an understanding of time. This understanding is connected to a measurable value, which is announced to us via clocks. However, this view of time was not always held and even today aspects of older time views are prevalent. A view of time that is context dependent is common to our speech patterns, where minutes can "feel like ages" and ages can "pass us by." The view of time is even further changed and developed in the computer scientist's view, from reduction of time being a stream of events to obsessing about the tick of a "clock."

This work is explicitly interested in how time is reasoned about, modelled, represented, and used. As a computer science work, there is a natural importance to the investigation of how time is used and represented in the field. However, this work is additionally, explicitly, interested in a representation of time that is more natural, particularly for non-computer scientists.

As such, the first section of this chapter will cover the philosophical view of time leading into the Computer Science view. The remaining sections of this chapter introduce topic areas of more in-depth interest in our context. Each area's representation and usage of time is explored in more detail. The coverage of these areas is only intended to cover the aspects of time, leaving details of the complete field for the reader.

3.1 Philosophical Views of Time

The study of time has been of deep interest in philosophy since the beginnings of such abstract endeavors. Covering the complete philosophical background in regards to time is a daunting task and beyond the scope of this work. Fortunately, works such as Schreiber's review [Sch94] can be leveraged. This section introduces the most pertinent ideas, deriving primarily from Schreiber. Starting with the philosophy of the ancients, the overview moves into the modern time, where the idea of time is highly influenced by the sciences, including Computer Science.

3.1.1 Early Philosophy

In early philosophy, space and time were recognized as differing from objects and processes. As Schreiber puts it, they were viewed either as:

> some kind of structured containers of events in the natural world (absolutist view) or just as some kind of abstraction to represent the relations among objects and processes (relational view). [Sch94]

Time was not considered an important factor in the philosophies, but rather just a tool of convenience.

The motion of the celestial beings was an important place, where time could not be ignored. Aristotle is accredited with being the first to note the interrelation of time and motion. Aristotle saw the need for them to define each other in some manner. According to Schreiber [Sch94], Aristotle thought of time "as the numerable aspect of motion."

As time began to achieve importance, modal logic became one of the cornerstones of the philosophical debate over time. Modal logics are concerned with what possible, probable, or necessary. The cornerstones of modal logic can be seen in our modern temporal quantifiers, e.g. always, sometimes, never, and before. Modal logic remains important today in various reasoning fields, as well as derivatives of it, like temporal logics.

By the 5th century, many of aspects of time that are today important were part of the philosophical view of time. At this point, the linear view of time solidified under St. Augustine as the standard approach. Augustine also brought the discussion of whether time is a "subjective feeling" or an absolute to the forefront of the discussion. According to Schreiber [Sch94], philosophy at this point was already concerned with many of the major issues:

- linear vs. circular,
- finite vs. infinite,
- openness vs. closure,
- discrete vs. continuity,

- absolute vs. relative ordering,
- objective vs. subjective, and
- definition of temporal modalities.

Many of these aspects will be handled again in Section 3.1.3, which looks at how time is structured and represented in computer science.

3.1.2 Physics

Perhaps the most important stage in the development of the philosophy of time is during the grounding of modern physics. It was during this era that time would take its place as a "first rate" quantity, as important as the three physical dimensions. The notion of time, as an independent variable, developed along with the field of Mechanics. This concept has taken root in every aspect of life; most people today see it as patently obvious that time is an independent variable, "the fourth dimension."

Mechanics introduced an absolute ordering on time. This view was challenged by Leibnitz with the introduction of relativism. Leibnitz believed that space and time only represent the relative order of things, a view that reaches back to the Modal Logics of the ancients. These two views were the main views of the times until both were, in some aspects, joined by modern physics, starting with Einstein. Einstein's relativity sets time dependent on perspective. Naturally, Einstein's concepts, while important, are in large part inconsequential for most VR applications, since travel over great distances are the one place people generally want to ignore accepted physics "limitations."

3.1.3 Structure and Representation of Time in Computers

As Allen noted in [All91], "One of the most crucial problems in any computer system that involves representing the world is the representation of time." Although Allen's focus is on modelling and testing real world problems in simulation, this statement true of any application of time performed in time. Basic literature dealing with the represent of time in Computer Science can be found in a selection of papers from the early 1990s [All91, Sch90, Sch94]. These papers form the basis of this sub-section.

The representation of time that is appropriate for a particular application is dependent on the problem being solved. Likewise, the time representation that is used highly influences the perception of the problem and the implementation of the system. A number of points are of high importance in this decision process. The most obvious to every Computer Science student is discrete time vs. continuous time; however, even this is a much more in-depth question than it seems at first. Adapting from Schreiber [Sch94], the main points of differentiation in the time representation can be identified as:

- time primitive,
- time topology, i.e. linear time vs. branching time vs. circular time,

- time extent, i.e. finite vs. infinite time,
- time density,
- orderedness of time,
- boundedness of time,
- causality of time, and
- directionality of time.

The most basic of these differentiations is in the time primitive used. In most Computer Science fields, this usually comes down to either *points in time* or *intervals*. Points in time, also called segments in time, most often correlate to discrete time systems, where time occurs at fixed intervals. This means that any changes in the system occur at only these points in time. A system built on points in time simplifies many tasks, such as the automatic verification of the system. When using intervals as the basic primitive, things happening over some span of time, are called *durations*. Intervals are defined as beginning and ending at specific points in time.

The concept of events is of importance in many areas and is related to the time primitive decision. The definition of events differs depending on the time primitive chosen. Events are sometimes seen as synonymous with points in time or time intervals, although most often events are considered to be discrete occurrences. When speaking of events, one says that it occurs. This speech pattern reveals a point that needs to be considered. A difference can be drawn between an event and an occurrence of that event. Following this logic, the event is a possibility of something happening and the occurrence is when the event actually happens. The event occurrence approach will be taken within this dissertation, as it provides clarity to the discussion in a simple way. Event systems can be programmed following either model.

Having a basic primitive of time, the next important aspect is establishing an ordering to time. Note that this implies that time is uni-directional, i.e. a time arrow. Humans are intimately familiar with a monotonic uni-directional time arrow, as we experience the world as such. Though this aspect is not addressed in the literature, as it has not generally an issue of importance, VR does not explicitly constrain time to be uni-directional. As with other things like storytelling, time is malleable in VR. Any reasoning on the system, the purpose of most of the time based research directions, is reliant on a time arrow and setting the ordering of time.

One of the most important aspects of establishing an ordering to time is dealing with how time's value is represented, the density of time. For points in time, developing an ordering is trivial. For intervals, ordering is more complicated. A relativist notion is most often used with intervals, establishing their order only with respect to each other. Figure 3.1 shows the possible relationships of intervals, as identified by Allen [All91].

There are three basic time topologies used: linear time, branching time, or circular time. Linear time is the classic concept from physics and is well understood. With linear time a *total ordering* to events can be established. In linear time, time progresses uniformly and constantly. Branching time, on the other hand, relaxes this constraint on

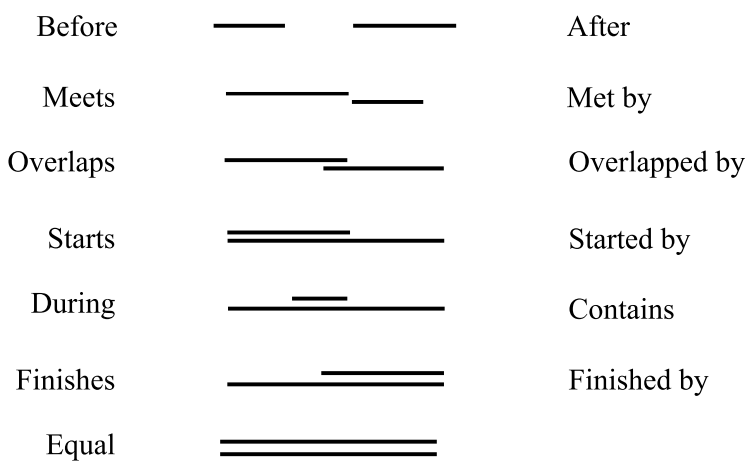


Figure 3.1: The 13 basic relations of interval algebra [All91].

the progression of time, allowing flexibility to the flow of time (though always forward). Branching time permits only a *partial ordering* of events. Branching time can be either left open, right open, or both open; right open is a very common view, as it means that the past is known and totally ordered, i.e. it is a linear arrow, and the future is unknown presenting various possibilities. This is often referred to as forward branching time. The final topology of concern is circular or periodic time. Circular time is used to represent recurrent events and processes. Circular and more complex time structures are only used as required by the system.

The structure of time, also referred to as time density, is concerned with the set of numbers that time's value is represented with. For continuous time, the set of reals, $\mathbb{R}$, is commonly used. Dense time is represented by the set of rationals, $\mathbb{Q}$. Discrete time uses integers, $\mathbb{Z}$, to represent time. This is essentially just a counter of steps. Along with all but the continuous representation of time, the famous "dividing instant" problem appears. The dividing instant problem deals with the question of what happens at the boundaries of the instant between two states. An in depth look at the dividing instant problem is provided in [MK03].

The issue of boundedness has already been seen in the branching time discussion above. For time in general, there is a question of whether there is a beginning and ending time. For intervals there is likewise the question of how the moments of starting and ending are defined. Closed boundedness includes the instant of starting/ending and is represented in mathematical syntax as [and]. Open boundedness does not include the very moment of the start/end, but everything greater/less than it. Generally, this means that a value is not defined at that moment. It is represented in mathematical syntax as (and). Finally, continuous bounding can exist, meaning values are defined for infinity, ∞, in that direction.

For interval based systems, boundedness is of utmost importance. This is particularly true for the "meets" relationship, see Figure 3.1. The first interval ends and the second interval starts at some instant. The intervals are separated by a dividing instant. The boundedness question here is which of the two defines the value at the moment. If both are open, the instant is undefined. If both are closed, then it is double defined. If one is closed and one is open, then the instant is well defined.

The computer has also brought a number of new views of time with it, although only to a small community. Computer hardware design is explicitly concerned with time. The clock signal in modern hardware is highly complicated and crucial to the proper functioning of the computer. Clock skews are a problem in today's micrometer processors, as things do not occur at the same time, when they should. Issues of concurrency and ensuring relative ordering determine the usability of processors. Multi-threaded and distributed computing have renewed interest in relativistic concepts. Ordering of execution for code segments in differing concurrent processes is of utter importance to the systems. This relativistic view is also critical for causal processes.

Another view of time that is largely due to computer science, dealing with time only via the clock. Time's progression is typically marked by a clock. Computer Science uses various clocks:

wall clock is the time in the physical world, i.e. what one sees on the clock mounted to the wall.

system clock is the ticks of the actual oscillator that drives the computer. Often this is the Unix system clock and is time, given in (micro) seconds since the "epoch."

execution time is the length of time that the program has been executing. This is typically reported as the number of cycles the program has been executed instead of in terms of the wall clock time, as multi-tasking systems cause the wall clock and executation time to be different.

simulation time is the current time in the simulation. This is current time in the simulated "world." It may be the time of day in the simulation or simply the time in seconds since the start of the simulation.

Typically the only time that is provided by the system, is the system clock which can be retrieved from the operating system. Other times are calculated by the program running.

3.2 Time in VEs

Virtual Environments are purely synthetic by their nature, and yet a majority of the VEs that have been created attempt to recreate reality. This is particularly true of their implementation of time. The time of a VE is almost always exactly in sync with the wall clock of the real world, holding the same properties that the mechanics

period developed. While this holds true for the majority of cases, recognition that the traditional limitations to time are not immutable can be found. Many of these cases were highlighted in the discussion of relevant VEs in Section 2.3.

Perhaps the easiest of the time changes in VEs is the "when" of an environment. In Digital Heritage applications, the VE represents a real place; however, the "when" of the place that is shown is generally from a different time than the current time. In all other respects the VE works generally tries to simulate the real world. The one case where this is not true is when the application slides the viewer through time to various points in time. Such a method allows the viewer to experience the real place as it changed over time. This is typically a discontinuous jump from one time to another, rather than speeding up time.

In the visualization community, there is also a common desire to look at things happening in time. While in some instances this is simply how things occur in "real-time," quite often the requirements are different. The rate of time's change may be desired to be faster or slower. This could be to get a better view of the process, for instance speeding up the erosion of a river bed or slowing down an explosion. In other simulations, the viewer may wish to speed through uninteresting parts and slow down the interesting parts (i.e. fast-forward/slow play). These methods are commonly called time squashing and stretching.

More common to simulations and VR is stopping time at a particular point. This may be done to investigate something more closely, but there are also more subtle reasons this is performed in VR. The first reason is that interaction in VR tends to be much slower than real-time. This is particularly true of interactions that involve more than simple selection of an object. The selection of an object is sometimes the second reason for stopping time. Selection of objects in interfaces is an interaction process of its own right, and particularly in the case of moving objects it can be very difficult. This topic is covered in more depth in Chapter 5.

A final point is that the VE does not have to be limited to a single time rate. Multiple times rates could concurrently be used with individual aspects of the VE being timed to one of the clocks. This is seen as an argument in the design of a few of the systems that are presented in 6.2. The author is, however, unaware of any actual applications of this possibility.

3.3 Computer Animation

The area of computer animation is naturally concerned with the flow of time. Computer animation is concerned with producing motion in components of the VE, both in real-time system and in offline rendering like movies. Because, animation is typically not concerned with interaction, the motion of bodies in time is defined by an "animator" a priori. The animation system is responsible for assuring that the predefined motion in time plays correctly. In a real-time rendering environment this is made more difficult, since the exact timing of the frames cannot be known beforehand.

Various books [JNW06, Par02, PR02, Vin03] covering the subject can be consulted for details on this area.

Several different methods of animation are used, though with respect to time these reduce to two main methods: key-frame animation and forward dynamics. Key-frame animation consists of a set of keys that define the position/orientation at a specific moment for a component of the animation. The timing of keys may be in equal steps or set to special points. At times between where the keys are defined at, the values are interpolated between the previous key and the next key. Key-frame animation is used for almost all "organics" in the scene and is typically created by an animator. Forward Dynamics are used for physical properties and are defined largely as they are described physics. As example of this is gravity on a falling body.

In attempting to be able to reuse animations and develop new animations, some research systems have been approaching time differently [BI06, WK88, WP95]. This is easiest to explain in terms of an example. Say that we have captured how several people walk, using some sort of motion capture system. We then want to extract the essence of walking, so that we can map walking onto any character, regardless of the character's size. Certain critical points of the walk can be identified, for instance the heel strike. Naturally these "key points" happen with different timing for each person. In order to extract the essence of walking, we can time shift the individual walks so that the key points all occur at the same time. A prototypical walk can then be extracted. In the inverse way, this prototypical walk can be time shifted to create an appropriate walk for any character. This idea can be expanded to include the possibility of creating different walks, where the timing is different, e.g. running and saunter.

3.4 Simulation

The simulation of systems is one of the oldest uses of computers. As such the area has well established method. Three major methods for the simulation of systems are used [PK05, ZPK00]. These methods differ in how time is handled. Continuous simulations allow states to change continuously over time. In contrast, the other two methods are discrete time systems. Classical *time driven* discrete simulation is built on a set time delta, Δt. Time progress strictly along the time axis in increments of Δt. *Event driven* discrete simulation is based on the timing of events in the system. At each step, the simulation moves time forward to the next event's occurrence and runs the effect of that event.

Time driven discrete simulation is the simplest of the three models in respect to how the system handles time. Time progresses linearly in the simulation by some predefined Δt. The state of the system may change on any step. The progression of the system is controlled by simple equations or even tables that determine what the state is at any time step. Such systems are particularly useful for simulations of systems that develop in discrete steps by nature, e.g. computer programs.

Event driven discrete simulation couples a time to the events of the system. The system is modelled by entities, representing components of the system being modelled. The entities are formed by their state, rules for their changes over time, and a time at which the next change occurs (the event). Entities register the times of their future events with the simulation kernel. This can either be done statically, setting the simulation events a priori, or events can be dynamically generated during execution. At each step of the simulation, the kernel jumps to the next time of an event. The entity is executed at that point, changing variables and in a dynamic system, potentially registering new events.

Several points are important to note in regard to this process. The state is only allowed to change through the events; therefore, all changes to the system must be represented as an event. Events can have any timing, i.e. they are not restricted to a time delta. This means that the system could be used to simulate continuous time. It is important here to note that simulation time is not related to "real time" or the wall clock in any way. Simulations of mere seconds may take hours to calculate. Simulations may consist of very dense calculations at one moment and then have no calculations for extended periods of time. This makes the method potentially ill suited for real-time constrained systems. A potentially interesting notion for DIVEs is the insertion of randomly timed events. Either at run-time or a priori, events that happen at random timing are introduced. In applications such as storytelling, such a functionality is likely to be necessary.

Continuous system simulation is based on a slightly different premise for modelling systems [ZPK00]. Differential equations form the basis of this method. The approach is sometimes even referred to by that name. The rate of change of the system is specified, instead of directly specifying the new values, as in the discrete time simulation case. The differential equations, often linear first order differential equations, must then be solved using numerical methods. Higher-order differential equations may also be used. Continuous system simulation is generally not real-time capable; however, this is because the systems are often very complex. One large advantage of this approach is that the time to calculate each step is of the same order or exactly the same (depending on the integration method used). This is highly beneficial for real-time interactive systems, as the flow of the system will remain consistent.

Interesting recent developments are hybrid continuous and discrete event systems. These systems build off the strong points from both systems and have potential use for creating DIVEs. The continuous system portion is constrained to piecewise continuous functions. These functions watch for thresholds to be reached on the real values and throw *state events*. The discrete event system includes these events, potentially changing the state values that the continuous system uses and restarts the function or can alternatively start a different function. A trivial example of this can be seen in Figure 3.2. *Time events* that are events based solely on the time can also be included. This makes a mechanism similar to random time event mechanism of event driven discrete event systems explained above. A useful resource for further reading on these hybrid simulation systems is Zeigler et al. [ZPK00].

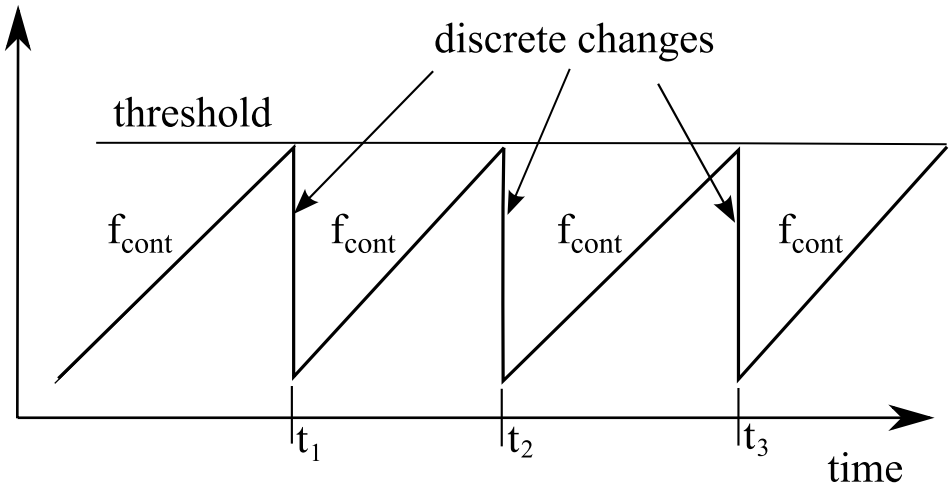


Figure 3.2: An example of a state event in a hybrid continuous plus discrete event simulation. After the continuous function passes the threshold, an event is triggered. That state event then causes a discrete event to change the value, instantaneously. The example is adapted from [ZPK00].

3.5 Reactive Systems

Reactive Systems are one of the three basic system types: **Transformational Systems**, **Interactive Systems**, and **Reactive Systems** [Sch04]. Transformational systems take inputs at the beginning of their execution, process that information, and return results at the end of their execution. Interactive systems are differentiated in that they take input during their execution, not only at the beginning of execution. Based on the action from the environment, the system reacts. Interactive systems and reactive systems differ in the reaction time required from the system. While interactive systems may take a period of time to respond to the external input, reactive systems have the requirement that they must react in "real-time." While Schneider makes the differentiation between interactive systems and reactive system in [Sch04], this difference is not often drawn. Within the context of Virtual Reality, however, this view of reactive systems is common, though rarely termed as such. Reactive systems may either be "soft" or "hard" *real-time systems* [KS05]. The other large difference between transformational systems and interactive/reactive systems is the length of time for which they run. Transformational Systems typically run for a deterministic period of time. In contrast, interactive/reactive systems are designed to run for a potentially indefinite period of time, reacting as input is received.

Many systems today fall under the umbrella of reactive systems, and it is an area of interest in the past decade [MP92, KS05, Sch04, Wie03]. The most prevalent are the ubiquitous embedded systems that run systems in everything from toasters to automobiles to nuclear power plants. These systems form the cornerstone of the reactive

systems research. The main interest of most of the research in this area is in modelling the systems and verifying that they model is valid. However, in various other areas of research, systems of interest are being developed that fall under the umbrella of reactive systems. Robotics are in almost all cases reactive systems and certainly real-time. Virtual Reality can also be viewed as a reactive system.

3.5.1 Modelling Reactive Systems

The modelling of reactive systems is performed using various methods. Most modelling is performed using Temporal Logics [MP92, Sch04]. Temporal Logics are extensions to standard Logics to handle time and timing aspects. Temporal Logics will be handled separately in the Model Checking section that follows. Temporal Logics are combined with a state modelling method, such as Petri Nets or Finite Automata, to create a representation of the system. Creating models of reactive systems is a large research area; two more practical explanations of the process and tools can be found in [Wie03] and [KS05].

In general the models have a number of aspects to them in common. The system can be seen as having two parts, the external environment and the internal computer system. The external environment is divided into discrete occurrences, typically referred to as *events*, and continuous aspects. While events follow the standard definition, the naming of the continuous aspects are various and conflicting between models. Depending on the modelling ideals chosen, the continuous nature of the system is sharply restricted, only being able to model linear systems or must be made discrete. In most cases, the continuous nature is defined by linear differential equations.

Regardless of how the modelling approach constrains the continuous nature of the model, their aspects are generally kept to one or two positions in the model. Discrete events, thrown either by external events or by temporal events, cause the state of the model to transition. Temporal events are either *absolute temporal events*, occurring at a specific global time, or *relative temporal events*, occurring at a specific time after a previous event.

Keeping track of time in the models is generally performed by *clocks*. The exact semantics of the clocks is dependent on the modelling approach. In the basic systems, the clocks are reset by events, so they count time since an event occurrence. In other systems, a global clock exists in addition to the standard clocks, counting the time since the model was started. Finally, a few systems include the ability to introduce clock skews, i.e. each clock can be linearly scaled.

Finally, the continuous portions can be brought in, if they exist. It is important to note that when continuous time portions are included in the model, the system is often classified as a Hybrid System, which is covered in Section 3.7. In most cases, the literature only suggests that the model includes *variables*, which are the continuously changing values. Usually, when it is explicitly said, the continuous values are changed inside a single state.

3.5.2 Verifying Reactive Systems

Having developed a model, the next phase of interest in the Reactive Systems community is *verification* of the model. Verification is important in many real-time embedded system application. Airplanes rely heavily on embedded control systems and the failure of the reactive system would lead to a catastrophe. Time plays a crucial role in the verification of the system. This is both true of maintaining the real-time constraints of the system and in maintaining safety and persistence properties of the system.

Verification of systems is performed through a number of methods. *Formal Verification* is the traditional method, stemming from the mathematics proofs tradition and is used with very high risks systems; however, this is very difficult and time consuming for Reactive Systems. Therefore, methods for automatically checking models have been developed. The most popular method for the verification of Reactive Systems is Model Checking [CGP99]. This method is most often based on Temporal Logics [CGP99, MP92, Sch04]. Model Checking and Temporal Logic approaches to time are presented in next the section.

3.6 Modeling Checking and Temporal Logic

Model Checking is one of the popular approaches of verification techniques for Reactive Systems [CGP99, PP06, Sch04]. It is an automated method for checking the model of the system. Model Checking verifies that a temporal logic formula and a set of all possible states of the model of the system form a valid representation of the system. However, this is only possible with a system composed of only a finite number of states. This limitation has significant impact on the usability of model checking techniques in our context. Interesting directions for DIVEs like emergent systems are explicitly excluded from these methods. More importantly, in order to ensure a finite number of states, the continuous nature of the system is often removed or severely limited. However, for VEs that require verification such a compromise might be required. The systems used in these endeavors, also have potential uses in our context.

Model checking, as with most verification techniques for reactive systems, builds on logics that are extended to handle temporal aspects. Temporal logics are the extension of other logic systems, often predicate logic, to handle time. These basic logics specify either state transition systems or *Kripke* structures [CGP99], to define the systems workings. Typically, for model checking, the temporal logics define predicates on the transitions between the states of the system. The most critically important aspect of temporal logics for our context is that they do not introduce time explicitly into the model, but instead only ensure ordering of events in time. Temporal Logics are in themselves a large field of study and starting points for learning can be found in any of [CGP99, MP92, PP06, Sch04].

Numerous temporal logics have been developed. CTL, LTL, and CTL* are among the most common temporal logics used in model checking. LTL is based on linear

time, and CTL is based on branching time. CTL* is a superset of CTL and LTL and describes properties of a computation tree. CTL* is composed of path quantifiers and temporal operators. The path quantifiers specify branching possibilities. The temporal operators present are [CGP99]:

- X ("next time")
- F ("eventually" or "in the future")
- G ("always" or "globally")
- U ("until")
- R ("release")

CTL* builds state formulas and path formulas using these operators. The formulas are true in a certain state and along a path, meeting their criterion respectively. The X, F, and G operators are straight forward in their meaning. The U and R operators combine two properties, such that one is true at some state and the other must be true at the states along the path, either before or after the state. The exact semantics can be inspected in chapter three of [CGP99].

Temporal logics such as CTL* are restrictive in their specification of time, making them unsuitable for many problems in our domain. However, specializations of these logics to include time explicitly exist. These extended logics are referred to as *Timed Temporal Logics* [PP06]. The Timed Computational Tree Logic (TCTL) is an extension to the CTL_{-x} version of CTL. TCTL uses time intervals for describing time restrictions on the formulas. The model of timed temporal logics can be either discrete (non-continuous) or dense; however, the discrete case is undecidable.

Penczeck and Półrola present the main points of timed temporal logics and describe model checking using them with *Timed Petri Nets* in [PP06]. Timed Petri Nets will be explored below. A similar approach can be taken by extending automaton to include timing information. These Timed Automata are introduced after the Petri Net extensions.

3.6.1 Time extensions to Petri Nets

A large number of extensions to Petri Nets that include time and timing information exist. The topic of Petri Nets and the time extensions of them are too large to cover here in detail. Books covering Petri Nets include [1]. The several surveys of the time extensions to Petri Nets exist [BY03, CR04, LNT00, PP06]. Here, we will give a short explanation of the major extension possibilities, without going into the details.

There are a number of aspects that differentiate the possible time extensions. The point at where time is introduced in the graph is the first major separation. Time is generally introduced to the *Places*, *Transitions*, or even occasionally the edges of the graph. This decision is not only conceptually important, but makes differences to the properties of the resultant graph.

[1] An introduction to Petri Nets can be found in [GV03].

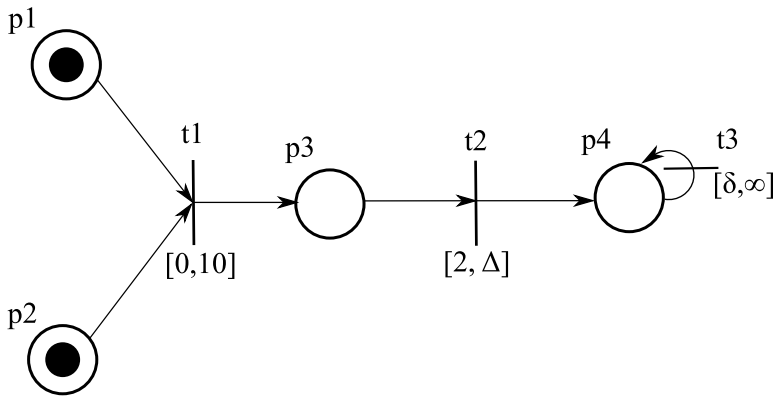


Figure 3.3: An example Time Petri Net. Places are represented with circles. Transitions are the vertical bars on the connecting lines between places. The solid circles found in p1 and p2 are Tokens.

The second point of differentiation to methods of introducing time is whether interval or discrete time is used. In the simple time case, a specific time or delay is specified that works on that node. For instance, a Place with a time would first "release" the tokens present to the transition after a delay of x seconds. With intervals, the specification becomes more complex, but also more flexible. Differing semantics can be used, but the general idea is to define a closed interval, or possibly right open interval, in which the constraint holds.

A third point of differentiation is on the *firing rules* that are associated with them. The firing rules control when the transitions are fired. According to Penczek and Półrola [PP06] there are three possible firing rules. The weak firing rule means that the timing is not determined. Strong earliest firing requires firing to occur as soon as it is enabled and the conditions are met. Strong latest firing allows it fire within a time interval, but no later than some time.

A final point is how the clock is modelled in the system. In Timed Colored Petri Nets, a global clock is introduced and the tokens get time stamps. Most extensions use a local clock on the extended entity (Place, Transition, or edge). Generally, the local clock is reset at some point in the running, for instance when a transition fires the local clock is reset. The exact timing and semantics of this can also vary slightly, changing the behavior of the model in certain instances.

The Petri net model can be realized either with a mathematical description or through a visual representation. Various visual representations of Petri Nets with timed components are used. An example of the visual representation of *Time Petri Nets* can be seen in Figure 3.3. The completed Petri net can then be check for various properties, such as: being deadlock free, boundedness, and reachability.

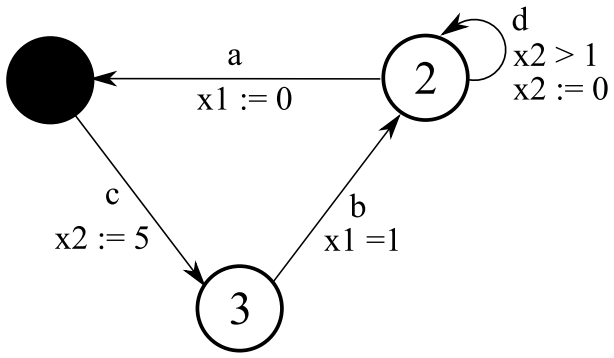


Figure 3.4: An example of a Timed Automata. The grayed circle is the current state. The timing constraints are shown on the transitions. Timing constraints are simple mathematical comparisons. Equations with := represent a change to the value when the transition fires.

3.6.2 Timed Automata

Timed Automata are a popularly used extension to finite automata[1] to include time [AD94, AD90, Alu99, PP06]. Alur, the originator of Timed Automata, says the concept "... provides a simple way to annotate state-transition graphs with timing constraints using finitely many real-valued clock variables" [Alu99]. In more lay terms, this means the transition from one state to the next can be constrained using timing information, where a constraint would be something like '< 3'.

This is achieved, more formally, by the creation of a finite number of clocks as part of the definition of the timed automaton. These clocks keep track of time. Additionally, on any transition, a clock can be reset to zero, thereafter counting the time since its last reset. At the beginning, the clocks can be initialized with specific values. *Clock constraints* are assigned along with the transitions and keep transitions from occurring when the constraints are not met. It is assumed that all clocks move forward at the same speed.

Transitions are assumed to be instantaneous. Time moves forward only while the automatons is in a location (state). Timing constraints are referred to as guards and restrict when the transition can take place based on the time of a clock. An example timed automata in diagram form is given in Figure 3.4. As a model checking technique, timed automata can be for verifying the system as usual.

3.7 Hybrid Systems

Hybrid Systems are systems composed of both discrete and continuous time components. The exact definition of them varies in the literature, from "discrete program

[1] Introductions to finite automata can be found in [HMU07, Sud06].

within an analog environment" [ACHH93] to a system truly composed of both aspects [Hen96]. Many of today's systems, particularly many of the embedded reactive systems, fall under the hybrid system classification. Among the most complex of such systems, are autonomous robots; they represent the continuous real environment with discretely sampled input devices and drive the robot's components with a combination of discrete output devices and continuous output devices.

Hybrid Systems research is, as with *Reactive Systems*, concerned primarily modelling and verification of the systems. What seems to be the most popular model and verification system is a generalization of Timed Automata, called Hybrid Automata. Hybrid Automata are more complex due to the addition of the continuous nature. This makes the systems much more difficult to analyze, as the state space potentially becomes infinite through the continuous functions. As such, the explanation of hybrid automata here builds on previous description in Section 3.6.2.

The Automaton are extended in a number of ways to include the continuous nature of the systems. State changes remain primarily as in Timed Automata, with constraints on any of the real valued variables. Instead of just "clocks," Hybrid Automaton can define a finite number of real valued *variables*. These variables hold the values that change continuously. Additionally, during state transitions the values can also instantaneously change value, so called "jumps." When the Hybrid Automaton is in a Location (state), differential equations that control the continuous nature of the system are active. In the original work from Alur et al., this was restricted to linear systems, i.e. limited to first order differential equations systems [ACHH93]. It seems this restriction is found in all systems, today. Transition constraints work as in Timed Automata, but may affect any variable. Additionally, the transitions (called jumps in some literature) can freely modify variable values in the instant of change.

In Hybrid Automata, all of the variables can be used as seen fit. This model allows a further aspect of potential interest. Using the continuous nature of functions the value of a clock can be skewed by a first order linear function.

In comparison to the discrete only systems found previously, hybrid systems cannot be effectively verified. Certain classes of hybrid systems can be verified, but in general the search space is not finite. One of the largest points of research seems to be reachability of systems, determining if states are even reachable. VR systems with user input are hybrid systems; this means that such techniques will be required for verification of systems if required.

3.8 Temporal Reasoning

Temporal reasoning has gained popularity and importance in the area of Artificial Intelligence (AI) in the last 20 years. Today, it is important in many AI related topics: robotics, planning systems, natural language processing, temporal databases, etc. The goal of these reasoning systems is to elicit information over the processes occurring

in some time dependant environment or data [FGV05]. Allen and Ferguson [Sto97, Chapter 7] identify three tasks of temporal reasoning:

Prediction what is likely to happen, given what is currently (and in the past) known about the environment

Planning how can a goal be achieved by the agent, given what is currently (and in the past) known about the environment.

Explanation determine (guess?) what is going on in the environment, given what is currently (and in the past) known about the environment. Alternatively, this could entail determining (explaining) the plan of another Agent in the environment, based on observations of that agent.

It is of importance to know what the term *agent* means in this AI context. While definitions vary, the main idea is that an agent is some entity acting in the environment in question. An agent could be a person, a robot, a program doing processes on a temporal database, etc.

The applications in AI using reasoning systems vary widely, from transformational programs to reactive systems. A large majority of the problems involve, however, real-time requirements. This real-time requirement marks a major departure from the modelling and verification of reactive systems described in Section 3.5.

In general, to reason about the processes that are occurring, the system relies on a temporal model and theory, the same as Reactive Systems. The two areas actually draw from the same initial work as basis of the temporal modelling languages. Of course, the major difference between the two area is that in a temporal reasoning system one is trying to have the program assert information about the temporal processes occurring instead of having humans define them as a basis for a new dynamic system, as is the case in for example Reactive Systems. As Grevini states in [Sto97], "The main reasoning tasks are concerned with determining the consistency of this collection (of qualitative or metric relations constraining time points or intervals), deducing new relations from those that are given and deriving an interpretation of the point(interval) variables involved that satisfy the constraints imposed by the give relations."

While the real-time requirements seem to make this potentially of more interest than the modelling and verification processes seen previously, the methods of temporal reasoning are not well suited to the needs of creating DIVEs. The area is interesting however, for the creation of one of the application areas of interest, Interactive Experiences. In some of the system ideas in this area, such as those presented in [BB05], the system should interpret information about the user.

One implementation method, however, has interesting properties for DIVEs. Because of the dynamic nature of understanding the world in real-time, the information over the conjectured or discovered processes must be a dynamic set. The set of data has to change as new data is taken from the environment. Because the timing of the information is critical to understanding it, tree structures are used to sort information

temporally. One such graph concept is the Dynamic Aspect Trees used in natural language processing, see [ter95] or [FGV05, Chapter 18]. The Dynamic Aspect Trees allow the tree of nodes to be modified at run time along with the temporal relations constraining the nodes. This is performed as new information is received during an on-going "live" processing. Such an implementation may be of interest for situations that involve highly dynamic worlds, such as Interactive Storytelling.

The reader interested in the temporal reasoning subject is suggested to start with a more general introduction to this complex topic, for instance in [Sto97]. The *Handbook of Temporal Reasoning in Artificial Intelligence* [FGV05] has a thorough coverage of the topic, but is much less accessible for the un-initiated.

Chapter 4

Dynamic Virtual Environments

The introduction of elements of change to Virtual Environments was identified in Chapter 2 as one of the primary methods to create more interesting environments. The basic idea of this was introduced in that chapter primarily through examples. In this chapter, a more formal analysis of these elements of change is presented. The elements of change that are of specific interest in this chapter are those that happen over time, *dynamics*. The definition of dynamics used is:

> *Dynamics* are anything that changes over time that affect perceivable changes to the VE either directly or indirectly.

This definition purposefully covers an immense spectrum. We feel anything that changes over time potentially makes the environment more lively and interesting.

The methodology that is taken in investigating dynamics was previously outlined in Section 2.5. The goal of this chapter is twofold: to gain an understanding of the design space of dynamics and to gain insight into how to best support the specification and implementation of dynamics. The initial step taken in this effort was to collect the possibilities for dynamics that form the design space. As previously alluded to, the design space of dynamics is expansive. Due to the sheer number of possibilities, a listing of the possibilities is not provided. Instead the design space will be presented in the form of a categorization.

This initial categorization structures the design space into an understandable form and provides insights to the manifold ways in which dynamics can be introduced to VEs. The categorization is developed from the viewer's perspective of the dynamics. The categorization is explicitly held separate from the implementation method, as the purpose is to define the design space that is available to the developer. The categorization is also expect to be the start of a design tool, in that it provides guidelines on what can be done to create interesting environments. This categorization will be presented in the following section.

A second categorization of the design space based on time provides insight into the requirements on a system to support Dynamics development. The time based

categorization is presented in Section 4.2. This categorization investigates how the dynamic is understood with respect to time. The categorization relies heavy on the understanding of time developed in the Time Chapter (Chapter 3).

Finally, Section 4.3 distills the implementation relevant portions of the first two sections to inform the design of a system of support for Dynamics in VEs. A set of requirements are extracted from the discussions of the classifications. Additionally, a set of suggested use cases that cover the most important Dynamics types is presented. These use cases are intended for use in considering the approach taken, as well as, for testing the usability of the developed solution.

4.1 Taxonomy of the Dynamics Design Space

The nature of and often the strength of Virtual Reality lies precisely in the fact that it is computer generated and nearly limitless in its possibilities. Although this is the case, the design space of the possible environments is only implicitly understood and no formal investigations of that space yet exist. This section undertakes an initial formal investigation of the design space of dynamics in VEs, via the methodology introduced in Section 2.5. As the design space is so large, the space is presented via the developed taxonomy of that design space. Even then, some portions of the taxonomy will have to be presented separately. The taxonomy is developed in respect to what the dynamic effects from the perspective of the viewer.

The taxonomy at the highest level can be seen in Figure 4.1. The dynamics design space can be broken into six main categories, from the most overarching to specific to most abstract:

Overall Presentation artifacts of the presentation method

Scene Attributes artifacts of the scene presented as a whole

Singular Objects changes affecting a single item in the scene

Propagating Quantities time changes effects that are not directly associated with the medium (object) in which they are present

System State artifacts of changes in the technology, either hardware or software, that is used in presenting the VE

Abstract Components intangible and conceptual changes that can only be indirectly experienced

Each of the categories represents a different direction that dynamics can take. The areas of Scene Attributes and Singular Objects are large in themselves and are only partially displayed in Figure 4.1. In the following section each of these areas is handled separately and more clearly delineated. In Section 4.1.2, the implications for system design based on the categorization are drawn.

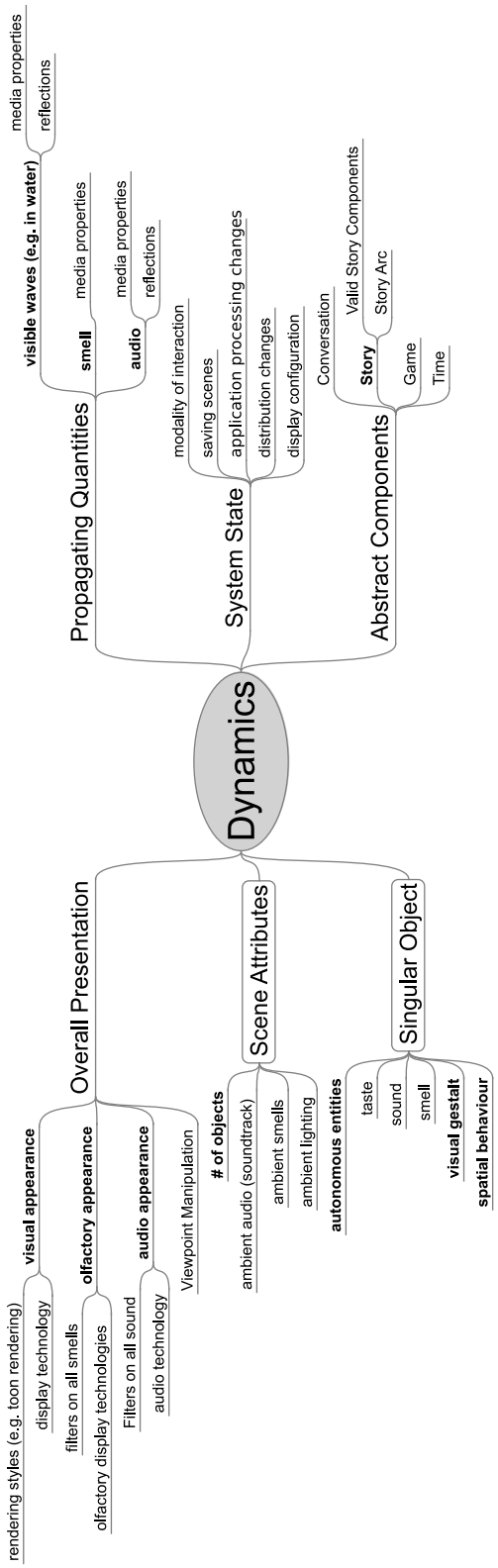


Figure 4.1: A mindmap categorization of the highest level of the dynamics taxonomy is shown. The sub-categories in bold print indicate the are further specified at a level below. In the cases of the Screen Attributes and Singular Object hierarchies, the complete tree is displayed in the coming development of those specific areas.

4.1.1 Dynamics Categories

Each of the six highest level categories identified are presented in higher detail here. They are presented in the same order as given above, most overarching to specific to abstract. The Scene Attributes and Singular Object categories are larger in scope and a further detailed taxonomy for those portions is presented.

Overall Presentation

The Overall Presentation category deals with artifacts present via the presentation method. These components affect the entire VE, but are differentiated from the Scene Attributes category in that they are not strictly part of the scene. Examples of these are filters that change how the scene is presented. For instance, visual effects could be added to change the appearance of the seen completely, perhaps (car)toon shading. Similarly, something as simple as the audio level fits in this category. A physical example of this is the warming of the projectors over time. This changes the color of the complete system (or worse, effecting portions of the display differently).

Another component that is vital to VR and related fields is travel through the environment. This is not a scene change, as the scene is simply there; instead, it is an outside factor. This viewpoint manipulation takes on a few different things. Head tracked users in VR are constantly in movement. Also, virtual travel is performed to move greater distances through the environment than allowed by the physical dimensions of the display used.

Dynamic Scene Attributes

A number of dynamics are present at the level of the scene created. These dynamics are part of the scene, but are not just singular objects within the scene. Figure 4.2 shows the scene attribute branch of the taxonomy as its own sub-taxonomy. This version is filled out at a higher level than previously presented in the high-level taxonomy. Each of the sub-groupings of this taxonomy will be addressed in turn.

Ambient effects across the different modalities fall into this classification. Ambient effects are, as the name indicates, those that are completely enveloping or surrounding. Ambient lighting is familiar to computer graphics knowledgeable readers. A common method of extending VE to make them more interesting is through the addition of ambient sounds, such as a "soundtrack." This is done as the familiar background film music. A similar method is having an unembodied narrator. In the case of the unembodied narrator, this point is included as the narrator is omni-present and is not attached to anything, but part of the scene itself. Note, an embodied narrator falls into the category of a singular object described next.

A common dynamic in VR based VEs is changes to the number of objects in the environment. This can either be the introduction of new objects or the deletion of objects. This falls in the scene attributes category because it is not a change of the object

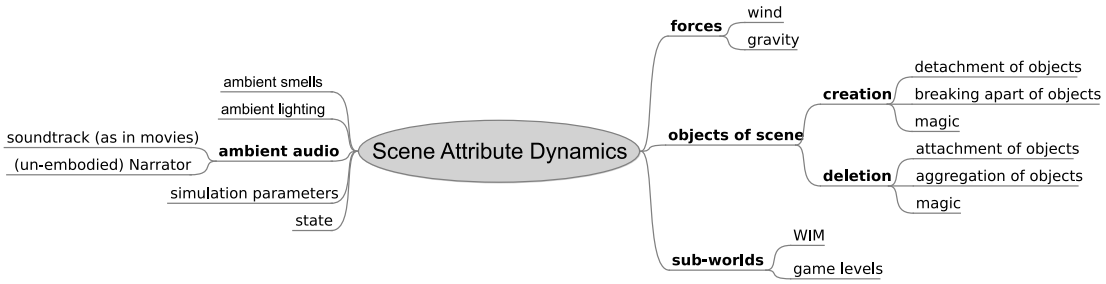


Figure 4.2: Taxonomy of the Dynamics that affect Scene Attributes.

itself, but rather a change of the scenes composure. Here we draw a semantic difference between objects that are present in the scene but invisible and the pure addition of new objects (and similarly that of objects becoming invisible). This case is handled in the Singular Object section. There are various reasons objects would be generated or removed. The most common is just having the ability to delete objects and magically make some object appear (usually via menus to select the object type. Another possibility is the breaking of an object into multiple objects, a conceptually simple, physically based phenomenon. Objects can also become attached to each other, for instance the lid of a thermos being attached to become a single object, i.e. aggregation.

A number of simulation and system related dynamics are possible. The addition of forces, including the notable gravity simulation, that take global effect is commonly a desirable addition to VEs. However, more generally this is simulation parameters and state values. These concepts are included here, but with the noted exception that they describe in general the implementation more than the concept understood. Forces, though invisible, are understood by the observer as an effect that is not part of the object. State changes and simulation parameter changes are not obvious, though they also should have effects on the environment. Any parameters that function similarly to those of forces at a scene level fit there. Parameters also exist at an object level.

The final changes that fall into this category are changes of the scene in its entirety. This typically reduces to a sub-worlds concept, as it is at some level still the same VE or at least the same program. Within the VR world, multi-application demo programs are commonplace. These are multiple independent applications that are combined into a single meta-construct. Perhaps more common is the level design common to computer games. Similarly interactive storytelling environments may have different scenes as the story progresses.

Dynamics of Singular Objects

A large subset of dynamics commonly thought of in regards to VEs is the dynamics of singular objects. This sub-category is the largest sub-section of the dynamics taxonomy. Figure 4.3, found on the next page, shows the mindmap sub-taxonomy of the dynamics

of a singular object. The two classical dynamics of virtual worlds are in the categories of *Spatial Behavior* and *Autonomous Entities* encapsulated. The spatial behavior of an object is the classical manipulation of VEs [BH99]. Autonomous entities included the classical virtual humans, which for many people is the quintessential dynamic of VEs.

The *Spatial Behavior* of the object concerns its movement through the VE. This is the position of the object in the scene as well as its orientation. Often scaling of the object is handled similarly. In respect to spatial behavior, the object is handled as a single item, regardless of its form. Changes to an objects form, those that affect the extent of the object, are considered elsewhere in the taxonomy. In computer graphics this correlates to the transformation of the object (in computer graphics lingo: via its center point). The object can either be translating or rotating of its own accord or as mentioned above, as part of user interaction. It is important to note that spatial behavior actually appears in other sub-branches of this taxonomy. In those cases the actual cause of the spatial behavior is important.

Autonomous Entities is the area that commonly comes to mind when considering the expansion of a VE to include dynamics, particularly for the uninitialized in the area. We divide the grouping into two sub-categories, *Agents* and *Simple Beings*. This separation is based on conceptual differences in the user's understanding of the entities. The difference lies in what the user expects of the entity. In simplest terms, depth of character differentiates the Agents from Simple Beings. This entails characteristics, like logic processing and emotions. It is important to note, that this difference is not necessarily human/non-human in a VE. These differences manifest themselves in the user's expectations of those entities. Users expect a lot more form Agent entities that Simple Beings. Both of these will be expanded on below and a theory based on this idea was introduced in [Blo07b].

Agents are epitomized by the Virtual Human, a field of research of its own [MTT04]. Agents or Virtual Humans are objects like any other and have the properties of objects, but because of the perceptions afforded by their presence (typically physical appearance that is human or humanized), the user expects intelligence and emotion that is present in a real human. In a distributed VE this may not be an issue, as the agent may be another human. For instance, in any of various modern Massively Multi-player Online Games (MMOGs), the other agents are simply embodied representations of another player. The generation of an agent by the computer, is generally very complex and beyond the scope of this work to list all the possibilities. Several of the possible parameters that can change over time are listed in the sub-tree. More to the research field can be found in [MTT04].

Simple Beings, while also autonomous, do not hold to the perception of having higher-level intelligence or emotions. These simple beings are a classical method in games to introduce autonomous creatures that seem to be intelligent. This could be the enemy combatants in a First Person Shooter. In the original concept created by Reynolds the entities were birds, something still commonly in VEs. Reynolds's renown algorithms was called Boids [Rey87, Rey00], a name still often used.

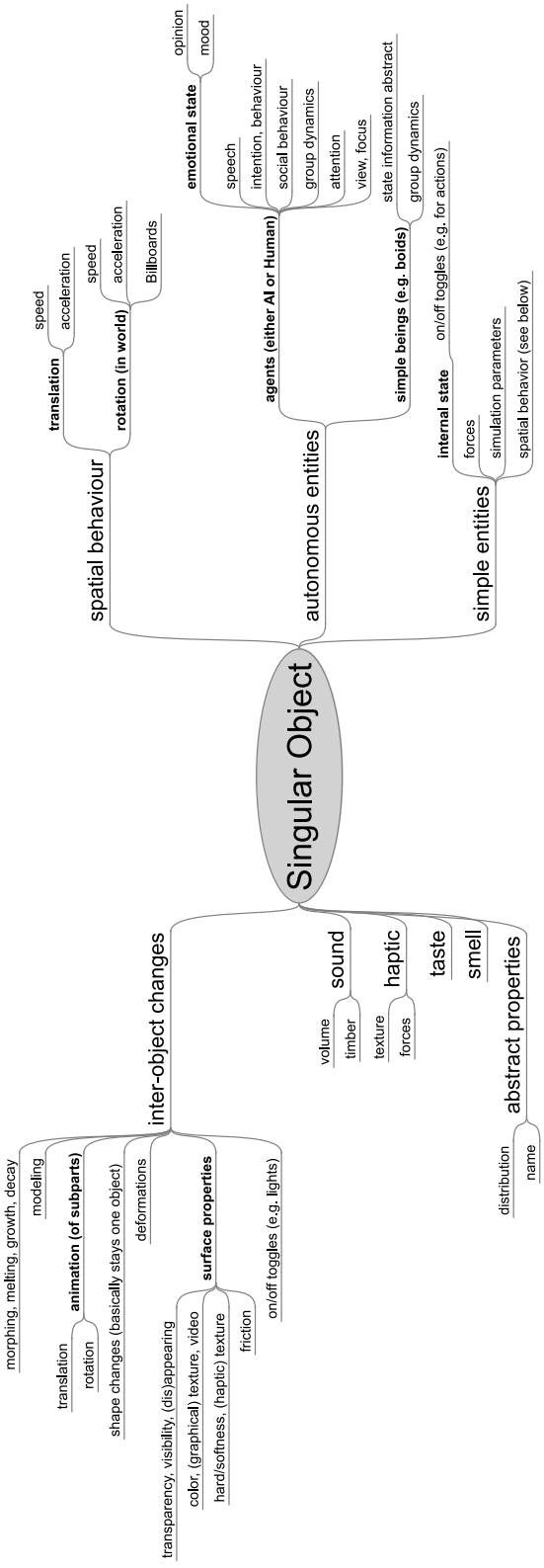


Figure 4.3: Taxonomy of Singular Object Dynamics

Boids have since grown into the more generically named "steering behaviors" area and is a very commonplace technique in the computer game industry for achieving entities with simple behaviors that show limited intelligence [Mil06]. The steering behavior and Boids concepts will be taken up again at the implementation level in the Boids example of Section 9.4. The difference to the autonomous entities category is that the user's expectation of simple beings is highly reduced. Neither higher-level cognition nor emotion is generally expected from the beings.

The steering behaviors methods used often introduce another interesting dynamic, group dynamics. This is a portion of how these algorithms create emergent behaviors. For instance in Reynolds original work, he showed the flocking of birds and herding of land animals. These group dynamics are intriguing to people. Group dynamics could of course, also be involved with either kind of entity.

Taking a step further in the reduction of intelligence, the *Simple Entities* category removes the expectation of intelligence from the object completely. Yet, the entity changes over time, typically a spatial movement. Examples of such objects abound in our everyday experience. These are typically objects that are acted on by some, often invisible, force that causes them to move. Examples from our everyday life include clocks or stop lights. The perception of the objects is that there is a reasonable explanation to the "why" question of its dynamic, typically explainable by physics or powered human design. The dynamics of the simple entity are not limited to spatial behaviors, but can include changes to the forces working over time or simple changes of internal state. Such state changes could even be the classical VE interaction of toggling the dynamics on/off. This would be the "light-switch" interaction that is often implemented for interaction with dynamic objects, a topic further investigated in Chapter 5.

As was mentioned previously, another dynamic could be changes to the extents of the object itself. Here, things like the animation of sub-parts of an object come into play. For instance, the rotation of the blades of a windmill would be a sub-part in motion instead of the object itself. Beyond sub-part motion, deformation of the object is a second and growing class of possibilities. This can simply be part of the animation of the object or for some other reason. Such deformations are very common to animation of characters. The skin deformation (a process called often called skinning) is an important and developing area of research in itself [GDCV99, JNW06, Par02].

Finally, each object can have changes across any of the modalities, e.g. the sound it produces changes as the machine gets rusty. An interesting possibility is the changing of surface properties, particularly haptic and frictions properties. In addition, an object could have abstract properties. These could include system attributes, like if the object is distributed or only locally available. Other abstract properties of an object could unlock a particular sequence in a story. Finally abstract information could be attached to the object, such as name, inventory number, etc. Certain applications may wish to establish this information at run-time, for instance in AR settings where the user labels items for the system to learn.

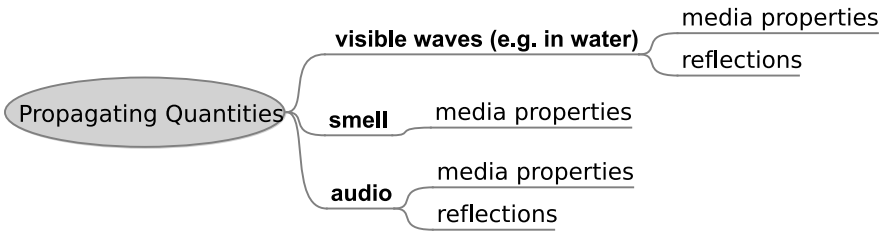


Figure 4.4: The sub-tree category of Propagating Quantities displayed on its own.

Propagating Quantities

On account of their unique nature, *Propagating Quantities* are in their own category. The branch is shown again in Figure 4.4. These propagating quantities are manifest only through another medium. Examples include waves in an ocean, waves in a pool of water, sound waves, and smells. They are unique in that they are to some degree independent of the medium in which they travel. In some cases, this is only a perceptual difference, but one that goes to the heart of the user's understanding. Waves move in the water medium, but they are perceived as waves, not as water moving. We feel this difference is potentially important. Another factor is the interaction these propagating quantities often have with the surroundings. For instance, take the classic pebble dropped in a pool of water. The propagation of the waves leads to reflections at the borders. Along this lines a more commonly implemented phenomena can be thought of, sound. The waves of a sound could only possibly belong to the "air." While the properties of the air affect the sound, it could be considered part of the air. However, sound is also very much affected by the surroundings, e.g. other objects. It could be alternatively argued that each individual case could be ordered under the scene attributes or singular objects categories. However, because the user understands the phenomena itself, these dynamics deserve their own category.

System State Dynamics

A number of potential places of change are not so much a part of the VE but rather of the system state. However, these impact the experience of a DIVE. Only one of these is a typical dynamic of existing applications, modal interaction. Interaction with VEs is often performed with only a single device, which is thereby overloaded in functionality. The changing between modes is an important dynamic of the system, particularly if a future goal is to simulate systems for purposes like quality control or proving the validity of a system. Related to this is the possibility of changing the display configuration or system configuration at run-time. In certain Scientific Visualization contexts this may mean changing the number of processors or even computers involved in the computation. Similarly, in distributed applications this can be changes the connections or changes to the aspects of the VE that are distributed [SZ99].

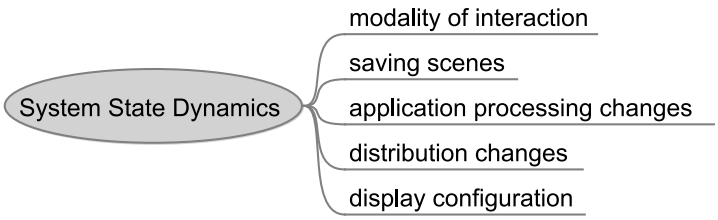


Figure 4.5: The sub-tree category of System State displayed on its own.

Another class of dynamics could be considered. Mechanisms such as saving models after in-world manipulations are performed at system level instead of directly affecting the VE. Since this actually enacts a operating system level actions that does not directly affect the VE (as opposed to a load action that falls under the scene level category by creating new objects), this qualifies a "system state" change. This is similar to Bowman's identically named portion of his interaction taxonomy [BH99, Bow99]. However, most of the state change interactions listed by Bowman create dynamics in other portions of our taxonomy. If we consider this as a form of persistence the question is whether this is simply an extended world dynamic. In cases like popular MMOGs this is more clearly the case as the world continues to exist at all times, with our without the user.

Abstract Components

Abstract Components of dynamic change are less tangible aspects, but aspects that are implicit and potentially powerful components to making a VE into an experience. Figure 4.6 shows this branch of the Dynamics taxonomy. Story and Game aspects of VEs are two of the major components at this level. Both of these necessarily unfold over time. In early games this was created primarily through levels, but today's attempts try to handle this just through a changing environment without the explicit changes. Interactive Storytelling environments are also implicitly dynamic VEs, particularly in that the stories are influenced by the user.

Conversations are abstract dynamics across various levels. The exchange of words embedded between characters of a VE, between viewers of a VE, between real participants in distributed VEs, and more philosophically, the artistic "conversation" as part of the environment could all be seen as potential dynamic components of a VE. Moreover, dialogue is necessarily an interesting turn-taking based dynamic, where meaning of any sentence is often fully dependent on the prior content of the conversation. Here we see a dynamic that is strictly ordered. A good introduction into dialogue systems can be found in [McT04]. A look at factors of dialogue in multimodal systems that are relevant to our use in VEs can be found in [MBD05].

The final major potential abstract dynamic of VE is time itself. As dynamics are defined as being aspects that change over time, this may seem intrinsically wrong.

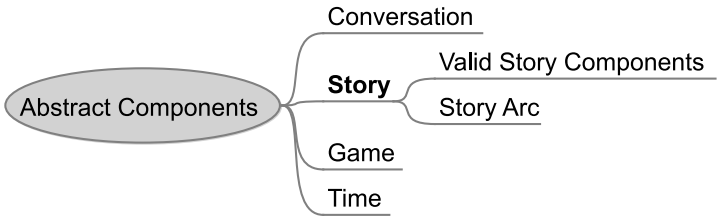


Figure 4.6: The sub-tree category of abstract components of dynamics displayed on its own.

However, in a synthetic world, time is not limited to moving in one direction or at a specific rate. More formally expressed, time does not have to be a homogeneous time arrow. Time in VR is flexible. This is a possibility that is often recognized and in simulations often desirable [Bry97, CHLB03, RJL+97, RLVD04]. Conceivable and even often suggested methods include:

- reversing time's flow,
- freezing time,
- changing time's "speed,"
- heterogeneous time skewing, and
- "undo" button for time.

Freezing time can be found in a few systems, specifically simulation based applications [Bry97, RLVD04]. It takes particular importance for interaction, as interaction techniques often are not real-time capable. Changing time's flow is a sort of grail among systems, often sought after but rarely delivered. Finally, the idea of having an undo for time is important, but ignored. Providing an undo functionality is generally considered necessary for good usability, but no solutions have yet been developed in respect to time-based interactions.

4.1.2 Implications

The dynamics categorization presented above provides a number of insights into the wide range of possible dynamics in VEs. Firstly, it shows just how many and diverse the possibilities are. More importantly, it provides a first look at what can be done in VEs to make them more interesting in terms of adding time dependent behaviors. Although this is not yet a guideline for such creation it provides a starting place for exploring the design space. In the design process it can be used as a tool for understanding how dynamics will be perceived and what effects they will have on the VE. Unfortunately, for the purpose of supporting the implementation, little can be learned from the taxonomy beyond a need to support different classes of dynamics. What we can draw directly from this, however, is a set of dynamics types that our supporting structure needs to address, i.e. use case sets. These can be derived directly as the category points.

Another observation is that multitudes of dynamics are introduced only through user interaction. This is interesting from the standpoint that it can be difficult to model a priori. The interaction that is performed makes a difference on how that dynamic can be modelled and how precise the complete modelling can be. Here, we can look to the simulation community, as noted in Section 3.4. The implication of this is that the dynamics and interaction system need to work together or even be completely joined. The topic of interaction and dynamics is covered in depth in Chapter 5.

One of the major benefits of this categorization is found when one looks at the dynamics found in the literature. In Figure 4.7 these typical dynamics are marked. This shows in a very explicit way how little of the design space is addressed. Although beyond the scope of this work, this provides a guide to investigating the full extent of the design space and the effectiveness of the respective dynamics.

4.2 Time-Based Categorization

The categorization presented previously was based on grouping of the possible dynamics by what effect they have on the VE. An understanding of the design space can been attained from that categorization, but little insight into implementation methods can be gained. A more informative classification to implementation needs can be created by investigating the differences of the main aspect of dynamics, time. An introduction to time was provided in Chapter 3. In this section, a categorization of the design space based on how the dynamics are represented in time is developed. After presenting this alternative taxonomy, the implication of this are discussed.

4.2.1 Categorization

Categorizing the Dynamics design space according to time could be done via various criteria. Section 3.1.3 presented a number of ways to do this. The most informative of these is to classify the design space based on the time representation that describes the Dynamic. What is important to note is that the concern here is the representation that best describes how the Dynamic is manifest and perceived. This is not necessarily how the Dynamic is implemented. This is important in context of Computer Science readers, who often hold fast to a belief that everything is discrete, since computers are discrete in nature.

The taxonomy built on this premise is shown in Figure 4.8. The entire design space is not listed, as it too large. Mostly, the categories are provided instead of concrete examples. In a few special cases we have included implementation details into the leaves of the taxonomy. In these cases they are prevalent implementations and very pervasive to the understanding of the dynamics in the eye of CG practitioners. For those practitioners, it is very difficult to separate that implementation understanding from the more abstract idea.

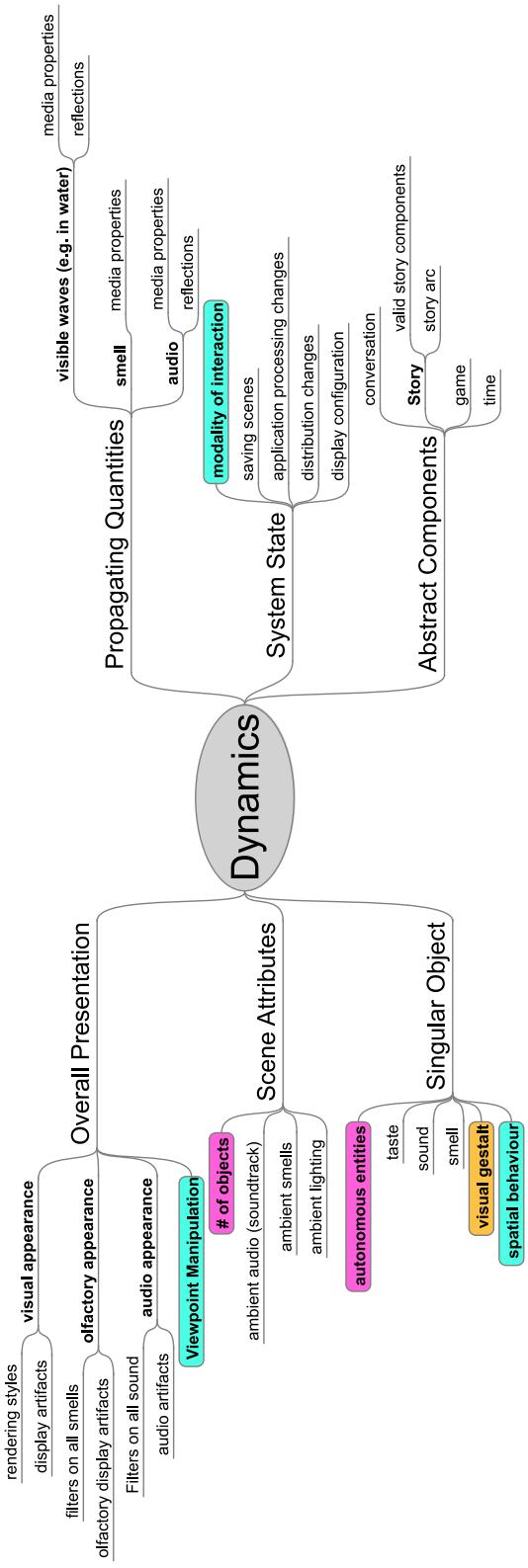


Figure 4.7: The Dynamics Taxonomy with the common Dynamics found in VR highlighted. The greenish colored items are dynamics that can often be found in VR. The magenta colored items are the next most likely Dynamics to be found, in a few applications. The orangish color item can found in specialized VR applications. Rare special instances exist for most all of the rest of the items.

The three main categories in the view of the world are: continuous, interval, and instantaneous. These categories define the time span for which the dynamics are defined. Also included is the boundedness of those time spans, which is important for understanding how to implement them.

In our definition, a continuous dynamic is defined from the starting instant of the application without end, i.e. $[-\infty$. Although we assert that it must only be defined from the starting instant forwards, the more general case is continuously defined which implies it is always defined, i.e. ∞. The most common way to define a continuous dynamic is a continuous function of some variable over time.

The interval dynamics are defined from the starting instant until some point in time later. Across the interval the Dynamics are continuous in nature. The length of interval dynamics may either be inherent to the definition of the dynamics or may be ended by a run-time determined instant. The differentiation with continuous dynamics is that they defined only over a finite length time, an interval. Intervals may in some cases be equal to the continuous category, excepting two differences. One is that the perceptual idea behind the Dynamic is finite rather than infinite. For instance, the perception of a physically based environment dictates the conservation of energy, e.g. the waves caused by a thrown pebble in water do not continue forever. Another difference is in interaction related elements. Take for example an object that is dynamic but has a possible interaction that stops its Dynamic. If that interaction is never performed the object behavior is continuous infinite. The next chapter will discuss interval dynamics in terms of interactions. Other concepts such as scenes, levels, stories, and conversations are finite in duration by their natures.

Instantaneous dynamics are discrete changes to some value that occur at some point in time. We will define them to take no time. Examples of this are typically found with respect to interaction, mostly correlating to state changes. For instance, turning on a light via a light switch in a VE is typically performed simply by selecting the light switch object. This causes both the modeled switch to jump (or possibly animate) to the "on" position and the light to suddenly shine. This happens in the instant of the selection. The creation/deletion of objects from the environment at run-time is another example, where in the moment the command is executed the object suddenly comes into existence.

A few distinct cases build an abstract of dynamic through a series of sub-dynamics. In all the cases we identified, the sub-dynamic was an interval based dynamic. What make the possible dynamic specific to this category is that each of the subparts contributes to something greater than themselves, i.e. there is some cohesiveness to them that makes them an entity on their own. The cases of stories, games, and conversation in particular could fall into this category. Each of these needs to be considered in turn, as their categorization here is each conditional.

Conversation, in a simple understanding, consists of a series of spoken exchanges, where each exchange occurs over a finite length of time. However, it may be argued that the moments between exchanges also convey content, either through silence or via

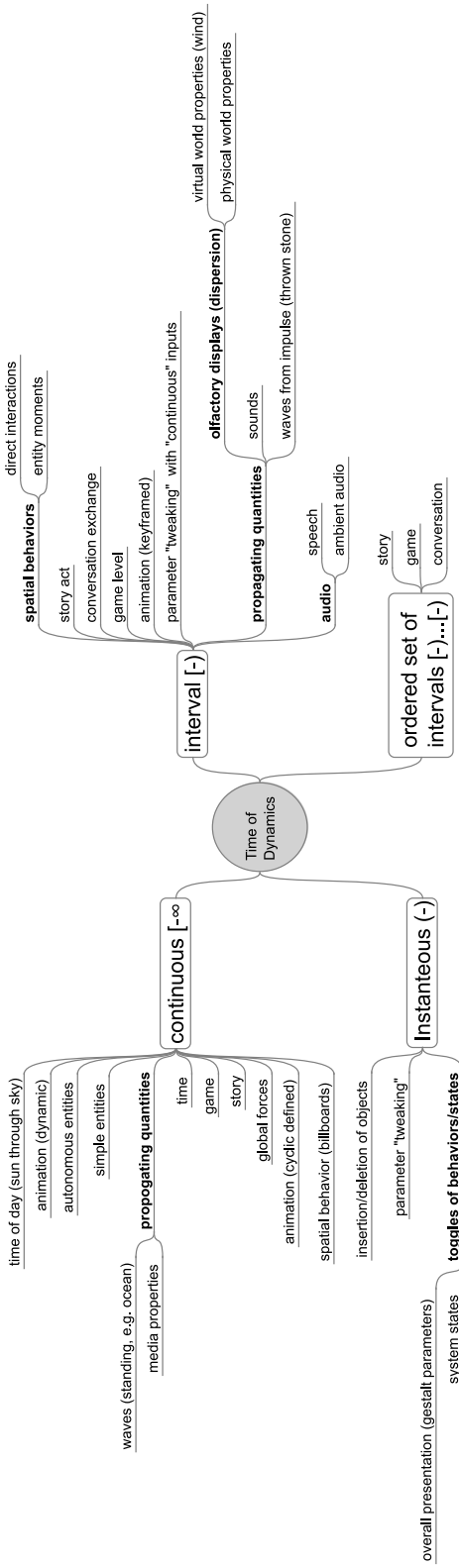


Figure 4.8: A categorization of Dynamics based on a time representation criteria is shown. The boundedness of the representation is also given.

non-verbal signs. This would seemingly make the complete conversation an interval in itself. For convenience, the conversation can be considered a set of adjoining exchanges, where the exchanges may be verbal or non-verbal.

Stories and games have traditionally been created as a series of acts (often called levels in games). Each act is defined over an interval. These cases are easily identifiable. However, modern variations strive to eliminate the classical abrupt change between acts, to the extent that it is no longer recognized. In those cases, they work on the user as if they are continuous. In VR settings, the transition to continuous stories can be important, since abrupt changes to the scene have been identified as contributors to disorientation and "cyber sickness."

4.2.2 Implications

The classification of dynamics with respect to time leads to a number of insights into the design space. Perhaps the most important observation is the distribution of dynamics in the categorization. Almost all dynamics are classified either as continuous or continuous over an interval. The impact of this is important to supporting dynamics. In Chapter 6, existing support systems in VR will be discussed. Only a few of those systems lend support to continuous dynamics programming.

Although not directly a part of the categorization, reflection on the nature of time emerges out of the considerations here. In the discussion of both time and its implementation in Computer Science in Chapter 3, many aspects of the representation of time were discussed. We find that many of these aspects are already found in the VE possibility. Because of the unique nature of VEs, there are several time phenomena that were not addressed in the prior discourse:

- there is not necessarily a "time arrow" in VR (time's flow is not established),
- time can be stopped,
- VR may have components that exhibit linear, branching, or circular time (or a combination there of), and
- time may not be homogeneous, but typically is.

These components of the nature of time in VEs need also to be considered in the design and implementations of supporting systems.

4.3 Requirements for Dynamics Support

The investigation of the design space of dynamics leads not only to a better understanding of the possibilities for DIVEs with respect to time based changes, but can provide insight into the development of a system to support their creation. From the initial categorization, a set of use cases can be developed to ensure that the system

provides the ability to support the entire design space. Drawing on the time based categorization, specific requirements for the system can be developed.

The categorization developed in Section 4.1 organizes the design space into groupings of similar dynamics. Those groupings show unique character, at least in how the dynamics are understood by the user. The categorization can be used to effectively define use cases for the system to be developed. The need for use cases as part of the method used for developing a system was discussed in Section 2.5. As such for the support of dynamics, the following use cases will be used:

Path Recording/Playback One often desired functionality is the recording and playback of paths taken through the environment. For instance, it is desirable for demonstration purposes as well as review of experiences in training applications and in therapy sessions.

Creation/Deletion of Objects in a Scene The creation and deletion of objects in the scene is a classical dynamic of VEs. Adding Support for it is more challenging than other simple state changes in the Scene Attribute category, since in its purest form it is a dynamic that requires run-time allocation/deletion of objects.

Spatial Behaviors The most basic dynamic of the visual environment is something moving spatial through the environment.

Inner-object Changes Showing that subparts of an object can be independently controlled from the object as a whole demonstrates the level of control of the system.

Entities The ability to create entities demonstrates a higher level of functionality and logic.

Story An abstract level of dynamic should be shown through a demonstration of storytelling possibility.

Time It should be shown that time itself can be manipulated in the VE

The use cases identified by no means ensure that all possibilities can be created, but are cases that are representative of the categories. In several cases, simple event triggered changes were avoided in favor of dynamics that are more difficult to create. In most of those cases the implementation is trivial in almost any system. In a few of those cases, the difficulty of the implementation lies not in triggering the change but in the implementation. An example of this is saving the scene. The use cases chosen strive instead to address the essence of each category.

A few requirements for the system can be derived from the analysis of the design space performed through the categorization work:

- support for time representations and implementation of:
  - continuous
  - continuous over interval
  - discrete time
  - ordered set of intervals
- support for sets that are order determined at run time
- support for run-time changes to the VE
- ability to manipulate time, including the ability to
  - freeze time
  - change itself, i.e. time's "speed"
  - define multiple time "speeds," potentially per dynamic
  - undo, including over all time representations (where possible)

A system which supports these requirements should be able to implement all of the test cases. The level of support for each of the time representations, specifically in terms of usability, needs to be addressed still via other factors, such as the developer's expertise and will be taken up again in Chapter 7, where a complete requirements list will be developed.

Chapter 5

Dynamics and Interaction in Combination

Dynamics and interaction were identified in Chapter 2 as two main ways to enhance Virtual Environments. They are the cornerstones to creating the kind of experiential environment we have termed Dynamic, Interactive Virtual Environments. This chapter explores the design spaces that are created at the conjunction of Dynamics and Interaction. Dynamics was already handled separated in the previous chapter. There, it was noted how many of the dynamic possibilities are created through interaction. Interaction has been a traditional focus of research. The standards of interaction within VR will be covered in Section 6.1.1.

As previously introduced in Chapter 2, Dynamics and Interaction can be combined in two main ways, as Dynamic Interaction and Interactive Dynamics. The difference between the two of these lies in where the emphasis is laid, either on Dynamics or Interaction. Dynamics Interaction deals with interactions that can be described as dynamics. Interactive Dynamics describes the case where Dynamics can be interacted with. Finally, there is a possibility of the combination of the two, what one could refer to as, Dynamic Interaction with Dynamics.

Each of these areas will be handled in turn in this Chapter. Although, it will be shown that the areas are not unknown to the community, they have yet to be formalized. The primary tasks of this chapter are to formalize the design spaces of these aspects of DIVEs and to derive requirements for a system to support their creation and the further exploration of the design spaces. Dynamic Interaction is introduced in Section 5.1. Interactive Dynamics are introduced in Section 5.2. Section 5.3 introduces Dynamic Interactions with Dynamics.

5.1 Dynamic Interaction

Many of the possible Dynamics in Virtual Environments identified in Chapter 4 were those induced by interactions. When we consider the interaction techniques that are typically used in VR environments, it turns out that most of them fall into the category of Dynamic Interactions. An example of this is the ubiquitous travel methods that are crucial to VR. Most of the methods used in immersive VR settings induce continuous dynamics, as the user moves through the environment. Strangely, Dynamic Interactions are rarely addressed with regard to their dynamic nature. To gain a better and more formal understanding of them, an investigation of them is undertaken here.

Given how common such interactions are, it is surprising that there is no term for such interactions. As such we are defining a new term to encapsulate this idea, *Dynamic Interaction*. We define it as:

> Dynamic Interaction is any interaction that either induces a Dynamic in the environment or the interaction takes place over a period of time. Interaction is here defined as the user taking influence over the environment.

To give a better sense of the kinds of interactions that fall into the category of Dynamic Interaction, Section 5.1.1 looks at the existing interactions of VR and related fields that fall under the categories umbra. Rather than being an exhaustive look, the purpose is to frame those interactions in a new light and to begin to give dimensions to the area. This is formalized in Section 5.1.2, where the design space of Dynamic Interactions is presented. An analysis of the design space leads to the categorization presented in Section 5.1.3. Section 5.1.4 presents existing approaches to the implementation of Dynamic Interactions. Finally, in Section 5.1.5 requirements for a supporting system are developed.

5.1.1 Dynamic Interaction in Classical VR

Classical VR includes interactions that belong to Dynamic Interaction. The pseudo standard classification of interactions in VR is Bowman's taxonomy. A more in-depth look at interaction in VR will be presented in Section 6.1.1. Of the four highest levels of Bowman's classification, three of them consist primarily of Dynamic Interactions. These categories are navigation, selection, and manipulation. With the exception of navigation, all components of what Bowman discusses on these topics are dynamic interactions. This section presents how classical VR interaction fits into the category of Dynamic Interactions. This will cover the interactions that Bowman identifies, but also include interactions that are more experimental and less commonly found.

When investigating the literature, an implicit understanding that the interactions are time dependent can be identified. For instance, there is a body of works about the effect of latency on different interactions [AHJ+01, ALE03, BKLP05]. In the coming paragraphs, we show this understanding across a number of areas, particularly those

where the time aspect is more explicitly represented. The discussion includes some information on implementations also, though that aspect will not be emphasized. Finally, the last paragraph will introduce the only work that truly addresses this time nature of dynamic interactions, a work on selection.

One of the classical issues in VR systems in terms of usability has been *system lag*. In a broader sense this can be termed temporal incompliance, as done in [BKLP05]. It has been shown in various studies that lag is one of the highest impact factors on interaction usability in VR, e.g. [AHJ+01, ALE03, WB94]. Although this is likely true of any interaction, in the VR context the studies typically deal with dynamic interactions. The most research aspect is the effect latency on head-tracked movement. Recent works with full citations include [AHJ+01, ALE03]. A single older work investigates the effect on hand-coupled manipulations of object has also explored [WB94].

Travel is the quintessential interaction of Virtual Reality. Travel is movement through the world that is controlled through software (as opposed to the tracked user's physical movement within the display space). Travel in VR is almost always performed as a dynamic interaction. Through some technique the user specifies a direction and speed of travel. A classic method is holding a pointing device and, when a button is pressed, moving in the direction specified. As with many cases, the dynamic nature of the interaction is only considered in that it is compared with the discrete alternative. For instance, for travel the discrete travel leads to jumps, which have been identified as a cause of "simulator sickness" [SS02].

The implementation of these travel methods typically involves defining some velocity at which the travel occurs. The time delta from the last frame is then retrieved (typically though system calls and saving time stamps) to find the actual movement each frame. One special case is however highly concerned with the time nature of the travel, "walk-throughs" or "fly-throughs." These two methods differ only in the number of degrees of freedom used in the travel, i.e. whether changes in height are allowed. The basic idea is to record the "path" taken and be able to replay it either later or in another session. Reasons for doing this include to create "canned demos" and to replay experiences for review purposes.

The fly-through is defined by following a path that is known a priori, but the implementation of this becomes explicitly concerned with time. In the case of replay, the exact same experience should be created. In a demo case, one of the motivating factors is to reduce actions that lead to "simulator sickness" that occur in live demos. The typical method is simply a key-framed animation for the "camera" path. Where the time nature very explicitly handled is when developing systems to record the path and in the playback. The timing of the playback has to match that of the original (if the keyframing is generated by hand, then such that no one gets sick). Typically, the major question usually resolves to what time scaling to use in capturing the keys. It is important to note, this does not actual capture the dynamic interaction, but instead only the effect of the dynamic.

A few of the less conventional interactions found in VR are naturally conscience of their dynamic nature. These interactions draw heavily on external areas. In most cases,

their integration into VR is either as an external module or performed by researchers from the respective fields in their own research. Notable areas that fit in this category are speech, gesture recognition, and haptics. In each of these areas the exact nature of the problem means that it occurs over time. These Dynamic Interactions are presented in the discussion of the design space in the next sub-section.

A special area of VR interaction work has dealt explicitly with the time nature. That body of work that deals with the input signals for the VR system, e.g. tracking data that gives the impulse to many of the dynamic interactions. For reasons ranging from removing device jitter to sensor fusion to prediction of values at a future time (i.e. closer to the actual draw time), filtering schemes on the input data have been used [RS01]. This is a highly specialized subset of research and the true aspect that deals with time is relegated to other research areas, namely digital signal processing. This means it is only relevant to the extent that there is a strong potential for the need to implement interaction within the system in order to support Dynamic Interactions.

Finally, a single work within the VR field truly analyzes the inherent time nature of the interactions [Ste06]. Steed's work looks at how to model the selection process with respect to time. Although a few specific interaction techniques have explicitly used time as part of the selection process, this is the only work to formally look at this nature. As the work is concerned with how to model and classify selection methods based on that model, it provides little input for the current task.

5.1.2 Design Space of Dynamic Interaction

The design space of Dynamic Interaction consists of various standard interactions and also interactions that are less conventional. A tendency to view interactions as only a result that happens at the end of the interaction coupled with the conventional way of speaking of the interactions means that they are often thought of as "simple interactions." A listing of the different types of interactions that make by the Dynamic Interaction design space is shown in Figure 5.1. It is shown in the familiar taxonomy more for consistency than the need for categorization.

Most of the standard VR interactions have previously be introduced. These are the first three of the main listings. Each of these consists of a large body of works and numerous different methods. The remaining possibilities will be described in turn.

Speech and gestures are two components that are still used mostly experimental in VR and are definitely dynamic interactions. By nature, they occur over time. In the case of speech, there are multiple levels of time related concepts to achieving understanding. Generally this involves, recognition of phenoms (the smallest part of a sound), words, grammar, sentences, and conversation contents. Up to the level of sentences that are all strung together the interaction can generally seen as complete, though from the implementation side words or even phenoms are complete [BSH08]. Similarly, gesture play out over time and have to be recognized via this time parameter. As singular interactions, speech and gesture based interactions usually are used to

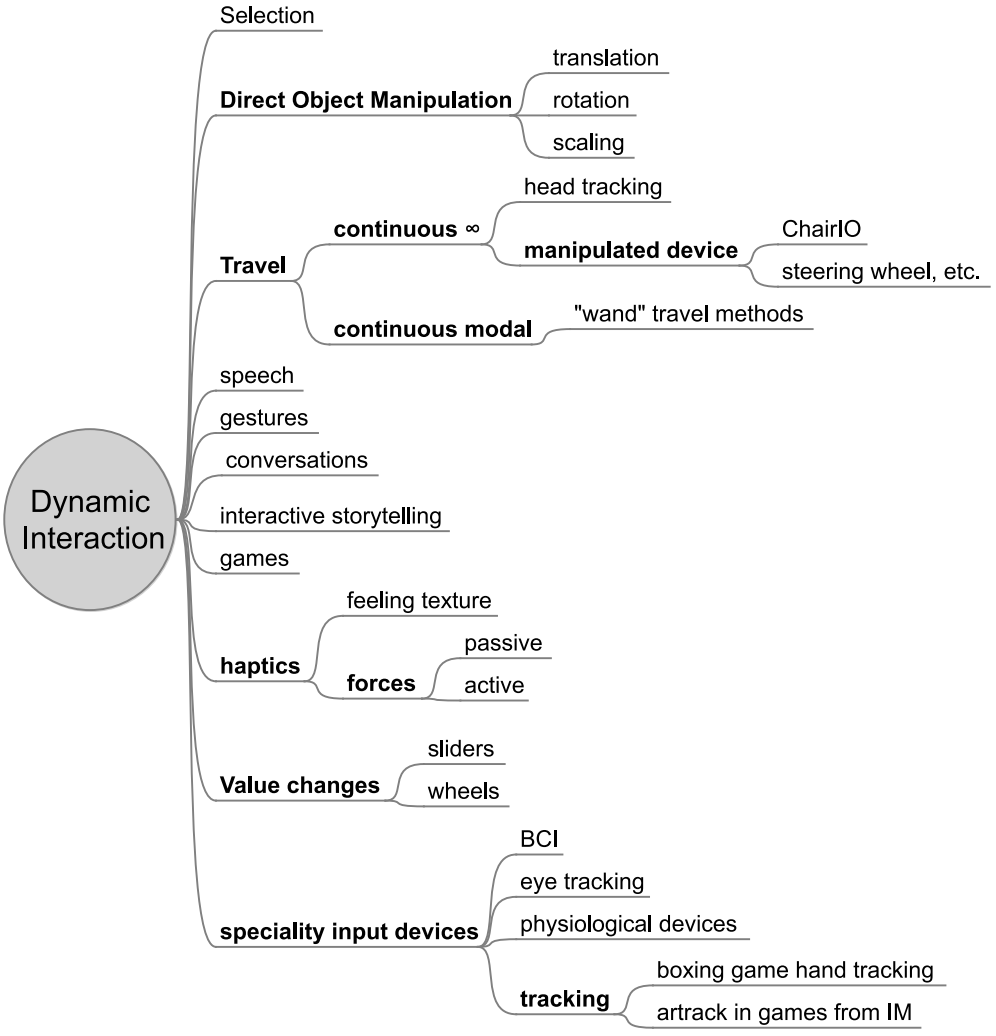


Figure 5.1: The design space of Dynamic Interactions is shown in taxonomy form.

launch single commands. Often this has been used as a replacement for menus, which are traditionally difficult in immersive VR settings.

Conversation differs from the other possible Dynamic Interactions in how it is handled over time [Vil03]. Conversation was discussed in the Dynamics chapter with respect to time and also in the Time chapter (Chapter 3. Conversation is per definition an interaction, as it has a "give and take" structure. The interaction in conversation is more abstract than the components that contribute to the conversation. In the simplest for it is just the exchange of spoken utterances. However, conversation is more about the meaning of the words than the words themselves. This means things like context play a part in the interaction, as well as, non-verbal cues. Determining the meaning of sentences, particularly with respect to time, is difficult. As can be seen, there are numerous time dependent portions of conversation as an interaction, part of what makes it complex to solve. Limited versions of conversation exist in some games, and the work of Pan et al. shows promise for its use [Pan07].

Similar to conversation, interactive storytelling and even games take input over time to reach the current stand of the story. Like conversation, the interactions play out a higher level and over longer periods of time. They are, essentially, the collection of all inputs and all of the run-time decisions of the system up to the current moment in the context they were made. Because they are high level structuring to the experience, they are often implemented as state machines. This works partially because the number of possibilities must be kept to a minimum. However, directions such as emergent stories make this somewhat obsolete as the story should develop on its own. For such environments, they are truly made up of all inputs, including dynamic inputs, and all run-time decisions. The realization of certain implementation details, particularly having an "undo" functionality, are impacted heavily by this.

Various input devices and their interaction fall under dynamic interaction. Head tracking, a hallmark of VR, is constantly changing input over time and was discussed in the previous sub-section. In addition to standard tracking devices, eye tracking technology is starting to become viable in VR. A few simple devices also exhibit a time based interaction, such as sliders and wheels. These devices are typically coupled into more complex interfaces. A few developing input devices also have continuous natures to them. A series of physiological interfaces allow the computer to react to the user in new ways [BK04]. Inputs like EEGs(brain activity), EKGs(heart measurements), MEGs(muscle activity), and skin conductance are taken continuously over time and often with respect to a user calibrated value that is taken earlier. Time adaptive algorithms are often used in evaluating the interfaces. Interesting on these interactions is that those inputs are often not directly user controlled, but rather give insight into emotional and unconscious reactions.

The final dynamic interaction category deals with haptics. Haptics can generally be divided into tactile and kinesthetic [BKLP05]. The tactile sense detects mechanical action (e.g. the texture of an object), temperature, and pain. Kinesthetics sense muscles contraction and relaxation, providing a rough idea of the forces on the body.

The proprioceptive sense is the third component of the human sensory-motoric system, and is the person's internal understanding of the position/angles of their body parts [BC03]. Tactile and kinesthetic senses are used as input channels in haptics. Proprioception is only passively exploited, but can have interesting effects.

Haptics are interesting from the time standpoint, as they all deal with time, but in subtle different ways. Although instantaneous forces are possible (or at least those so short that they are less than a time of a single frame), forces are per definition related to time ($f = ma$, where a is acceleration a time dependent term). Generally, in haptics this means a motor applying a force to a device coupled to the person. This is an interaction, because the user has either will be moved or applies an opposing force. Texture deals with portraying the surface properties of an object to the user. Texture requires having different forces working over time and space. Texture is not directly generated, but rather the haptic output provides a sensation that is perceived as the texture desired. Proprioception can be exploited to trick the user's perception of their actions. For instance a using a method called "redirected walking," one can make the user walk in circles, while they believe they have walked straight [RKW01]. This is done so that the user can physically walk through a much larger virtual space than the physical area. For such perceptual tricks to work, they generally have to accumulate over time.

5.1.3 A Time-based Taxonomy of Dynamic Interactions

The classification of Dynamic Interactions by their respective time properties again produces insight into design space. As with dynamics, categorization in this respect has implications on the implementation of the interactions. Figure 5.2 displays the design space categorized again by the time representation require and its boundedness in time.

When categorizing based on time representation, the dynamic interaction design space is divided into four categories. Two categorizes are continuous in nature, differing by their boundedness in time. These two categorizes are continuous infinite and continuous over an interval. The second set of interactions involves ordered sets of interactions that occur over time. Here we differentiate between ordered sets that are composed of either continuous over intervals segments or discrete events.

The boundedness difference for the continuous time categories may seem trivial and irrelevant. However, this may prove to have an impact on the implementation, making it more than a conceptual difference. In the continuous infinite case, we expect the interaction to be active over the entire course of the application's life. A classical example of this is head tracking. Tracking of an input device used for travel, though, typically falls under the realm of continuous interval, as it almost exclusively does as a modal interaction.

Outside of a head related view transform, almost all interactions are performed modally, giving them an interval extent. Interactions performed continuously over the

entire period of the application can occasionally be found, but typically are disregarded in order to give the user a break. A few special cases can be found, particularly in the realm of travel. Examples of this are walking interfaces (both treadmill like devices and "walk in place" methods) and other specialized interface, like the ChairIO [BBH05, BBH07]. The ChairIO is a travel interface based on a special stool. The ChairIO is unique in that it is a non-walking interface that allows such continuous travel input without fatiguing the user (mentally or physically).

The second set of categories base on the premise that the interaction is compose of an ordered set of interactions that are distributed in time. The individual interactions take meaning as an interaction through their association together. In this sense these are interactions that occur over time. A now familiar example is that of conversation; it is a collection of speeches that occur in a specific order. The possibilities identified have already been discussed in the prior sub-section.

Games and Stories can be found as both sets of intervals and of discrete events. Although this deals primarily with implementation details, there is a conceptual impact on how the interaction is viewed embedded in making the difference. Classically, implementations view the interactions as a collection of discrete occurrences. This is partially because all interactions in storytelling and games have been seen implemented as discrete interactions, but mostly because the implementation is based on a state machine approach. For instance, all of the things the user did in the game reduce to steps, like user opened the door and user killed the bad monster. While this is seen this way and the story of the experience is retold this way, the individual components of the set may be continuous over an interval. For instance, the user may be required to water a plant over time in a story in order to move one. Here it is not a single interaction that causes the story to advance, but rather a series of interactions (that are themselves a set of discrete interactions) over time.

5.1.4 Implementing Dynamic Interactions

Many of the identified Dynamic Interactions have been identified as commonly found interaction in VR or are part of other areas of research being incorporated into VR. This sub-section briefly introduces the methods already being used to implement the various possible interactions not already explained as part of the clarifications in previous sections. Informed by this, requirements for support are developed in the next sub-section.

Direct manipulation of static objects is possibly the easiest dynamic interaction to implement, and it is already commonplace. There are various advanced methods, but in general during the dynamic interaction, changes to the input result in a directly derived movement to the object being manipulated. For instance, with standard ray picking described in Section 6.1.1 the standard translation manipulation occurs during the time a button is pressed. In that time, any movement the user makes of the picking device moves the object so that it is the same distance from the user, but re-positioned

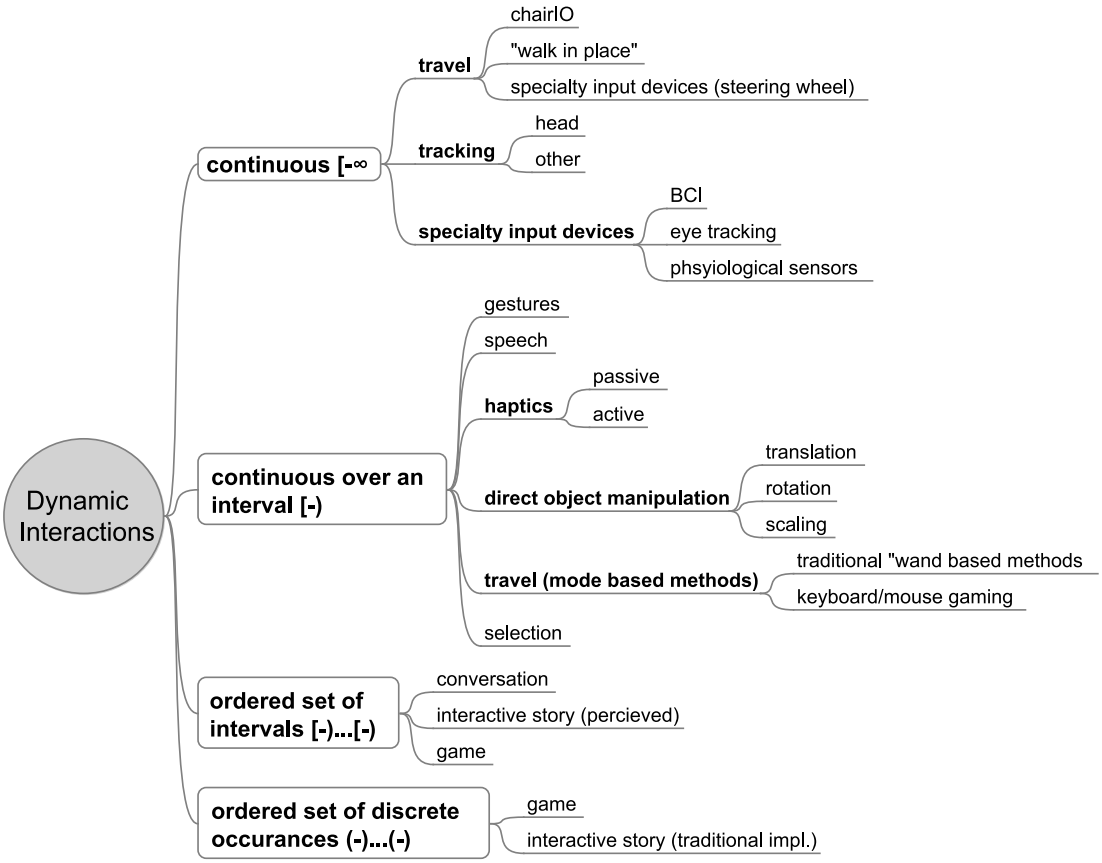


Figure 5.2: A time based taxonomy of Dynamic Interactions

to be where the device is currently pointing. In [AHKV04], they even refer to methods using this principle for navigation "continuous wand model," a nod to this nature.

Many of the advanced manipulation techniques address predominately the difficulties that occur in manipulation of objects that are distant from the user. In a few very specific cases, time is factored into the direct manipulation method. For instance, some control/display ratio algorithms factor in how quickly the user is moving the object. When the user is moving slowly, it is assumed they are trying to be precise and the C:D ratio is made slower, such that the movement of the object is smaller than the movement of the user [FKK07]. This requires not only time, but the velocity of the movement to be determined.

A potential time related problem with interactions is temporal incongruities of data due to the typical VR system structure, as explained in Section 6.1.3. Because the input devices, and in particular the tracking equipment, run at different rates as the VR system control loop, the VR system typically samples the inputs at the beginning of the "application" part of the loop. All of the interactions that are calculated are dependent

on that value. The latency in such sampling is not the only potential problem of this standard method for dynamic interactions. This method typically, just ignores all values available, but the most recent. Depending on the dynamic interaction implementation desired this may be undesirable, for instance if the interaction is dependent on the velocity of the interaction at the moment of interaction. In cases like this the "jitter" inherit in the tracking system (and the user's hand) is amplified, so filtering is necessary. Another potential problem is sensor fusion, since the inputs all were taken at slightly different times. Interaction systems like openTracker [RS01] help in the creation of such filter systems, but are non-standard.

Gesture recognition is any input, where the pattern of the input over time defines an action. Although, not innate to gestures, the most common implementation is to have completed gestures generate an event or call a specific function. Symbology, such as that used in sign languages, is used to determine the meaning. The mapping from a time based pattern to a single discrete input, can be found in various methods addressed in this chapter. Alternatively, gestures could be used to create Dynamics. Existing examples in real contexts are rare, but the famous "Minority Report" movie scene is a prime example of what could be done. It is important to note the difference between gestures and postures. Holding a static pose is a posture. Gestures are dynamic by nature. In some cases they are a timed transition through a series of postures. An overview of gesture recognition in VR can be found in [Tur01]. A more hands on starting point is the Georgia Tech Gesture Toolkit [WBAS03], GT^{2}K, which adapts a speech recognition toolkit to perform gesture recognition and is openly available.

Speech recognition is also a dynamic interaction, similar in nature to gesture recognition. Speech recognition is not typically seen as a research area in VR, but has been used in VR. Starting points for implementing speech technologies include books like [SQS04] and [Raj07]. Again, the most common used method in VR is the launching of events via single word commands.

Turn taking in conversations is an interaction that builds on speech. Conversation is an emerging necessity in worlds that involve interaction with avatars. An overview of the complexities of human conversation can be found in Vilhjálmsson's dissertation [Vil03]. Within the temporal realm, a typical problem is that of "turn taking" in the conversation. The system has to detect and even anticipate the speech of the user and not just speak over the top of the user. There are some indications, however, that failure to achieve this may not always be critical. In a study from Xueni Pan [Pan07], it was seen that improper timing of the virtual avatar did not seem to affect the user's level of presence in a stressful situation.

For many of the higher-level interactions, such as conversations, the area of temporal reasoning, introduced briefly in Section 3.8, is relevant. This area of research focuses explicitly in inferring information from what has previous happened over time. Temporal reasoning is also potentially relevant in advanced storytelling and gaming concepts, although the author is not aware of any applications of those approaches to storytelling or games. Temporal reasoning could be used to determine the user's

strategies and react to them or to be able to deduct behavioral changes in the user too. Related approaches have been taken in the virtual human area, specifically in the logic of game opponents.

5.1.5 Requirements

Drawing on the information presented in the previous sub-sections over the design space of Dynamic Interactions, a set of requirements is developed. These requirements focus on the components necessary specifically for the development and integration of Dynamic Interactions in a new support system. The requirements identified are:

- support for time representations and implementation of:
  - continuous
  - continuous over interval
  - ordered set of (continuous) intervals
  - ordered set of discrete events,
- enable handling of input
  - including all values, not only the most recent
  - including support for writing filters, in particular time based ones
  - supporting direct manipulation interactions,
- support for inferring higher-order functions of time on continuous inputs, e.g. velocity and acceleration,
- support for multiple output modalities, with different update frequencies,
- support for hierarchical levels of Dynamics Interactions, for instance for conversation, which is respectively composed of speech interactions,
- provide an ability to record and play back Dynamic Interactions, and
- provide an ability to undo Dynamic Interactions.

A further consideration that may not qualify as a requirement of the system, but should inform the design, is that a number of implementation directions noted already have developed methods and implementations. These implementations are already well developed and hopefully already implemented in appropriate manners for their task. The ability to leverage those existing systems in conjunction with the developments here should be heavily weighted over redeveloping those ideas in this framework.

Use cases for Dynamic Interaction are fairly straight forward:

Direct Manipulation Continuous direct manipulation of objects should be demonstrated.

Higher-order Direct Manipulation The continuous direct manipulation of an object, such that the system derives the velocity of the user input as part of the interactive nature of the system should be demonstrated.

5.2 Interactive Dynamics

Interacting with Dynamics is an interesting direction of consideration, when Dynamics are added to the VE. These interactions form what is referred to in this work as Interactive Dynamics. We define the term as:

> Interactive Dynamics are any interaction, where the "object" of interaction is a Dynamic. Interaction is here defined as the user taking influence over the environment. Dynamics are parts of the VE or its control structure that change over time.

Dynamics are precisely those discussed in Chapter 4. The subset of interactions here are those that are with Dynamics that are changing at the moment of interaction. In the definition "object" is specially marked. This is to indicate that the "object" of interaction may not actually be a visible object of the VE. In Chapter 4, Dynamics that were not virtual objects themselves were identified and could be what the user interacts with. For lack of a better encompassing term, we use object here.

Interactions with dynamic objects are performed in real life, though only a few cases are frequent for the average person. Many sports involve such interactions at the core of the change, for instance kicking a moving ball. While not every actively plays such sports, the "simple" act of having a conversation contains exclusively interactions that fall into this category. Section 5.2.1 provides longer descriptions of several types of Interactive Dynamics found in VR.

We believe Interactive Dynamics are important in VR settings for an additional reason. Although not yet formally studied, anecdotal evidence indicates that user have much higher expectations of interaction with dynamics in a virtual world than the real world. Even in high presence environments were the users react to situations as if they are real, users often try to interact in ways and with things that are obvious not possible in the real world. An example is the desire to interact with a massive stone pendulum in the "Cuevo de Feugo" (the Fire Cave) [WELCN97]. This reaction is often less than a minute after the same people ducked so that they would not be "hit" by same object. This heightened expectation of interaction with objects makes it critical as VEs become more dynamic and interesting. Enabling those interactions is also a way to move toward the creation of experiential VEs.

In order to provide a more concrete idea of what Interactive Dynamics consist of, a series of examples drawn from VE communities are provided. The design space of Interactive Dynamic is then formalized in Section 5.2.2. Section 5.1.4 discusses some of the existing implementation methods that are relevant for the creation of such interactions. Finally, a set of requirements are developed for supporting the creation Interactive Dynamics and to enable exploration of this challenging area.

5.2.1 Illustrative Examples

In this subsection, a number of different existing Interactive Dynamics are discussed. The examples highlight the different possibilities for Interactive Dynamics and provide an insight into the significance of them. Each example highlights a different type of Interactive Dynamic. In some cases, the technical side drives the method of interaction.

Discrete Interactions

Discrete interactions with dynamics are the simplest approach that can be taken. As such, there are more examples to find; however, the examples are not significantly different as the interaction performed reduces to a single "event" that is launched as interaction is initiated. A well widespread example is the action of "First Person Shooters," also known as "Ego Shooters." While in a few rare cases a "flight time" is incorporated with special weapons, the typical shooting game has an instantaneous shot. In these games the interaction is causing damage to an opponent or object. In a more general sense, the interaction is any discrete change to the object. For instance the color of the object could be changed, for instance red as a "highlighting" method.

Crayoland

One of the more interesting VEs on many levels, is Crayoland, the first VE for the CAVE™ [Pap, Pap98]. Crayoland is a simplistic virtual world, with very few polygons. An image of Crayoland can be seen in Figure 5.3. It derives its name from the fact that everything is textured with hand colored images (using Crayola Crayons™). However, Crayoland is still relevant, as it is still used in various locations as the standard demo and various VR researchers continue find it to be the best VE. Among other interesting aspects, Crayoland has two interactive dynamics, a butterfly and a colony of bees.

In a portion of the grassy land, a butterfly flies around. The user has the ability to interact with the butterfly, via their virtual hand representation. If the user holds the virtual hand still in the vicinity of the butterfly, it lands on the user's hand. The butterfly sits on the user's hand and waits, as seen in Figure 5.3. When the user makes a large movement, the butterfly flies away. Prior to landing, the butterfly is a "simple being" with a spatial dynamic. The interaction initiated by the virtual hand's vicinity and its being held still causes the butterfly to "attach" itself to the virtual hand (creating a Dynamic Interaction).

The second dynamic involves a group of bees found further along the grassy flats. The bees of the world spend their time either at the hive or collecting pollen at the flowers, without regard to the user. However, if the user swats the beehive with their virtual hand, the bees become agitated. They swarm around the user, using an adapted Boids type algorithm, i.e. a simple being in our dynamics classification. This example illustrates how such simple beings are often an Interactive Dynamic; the bees fly around, but their position is dependent on the position of the user, such that they follow the user.

Figure 5.3: An image of the Crayoland VE for the first IPT. In approximately the middle of the screen a colorful butterfly is perched on the virtual hand (the gray blob) of the user.

Additionally, the group of bees exhibit a group dynamic, a form of Interactive Dynamic, within themselves; the bees react to the position of their neighbors, so that they don't collide with each other. This viewpoint requires considering that the interaction of a virtual object with respect to another.

Classical Avatar Interaction

Classical avatar interaction in areas such as games is an interesting area for Interactive Dynamics. Interactions with avatars show multiple levels of Interactive Dynamics. Generally such interactions center on conversing with the avatar. This is typically a one-on-one conversation. Conversation in itself is one of our Interactive Dynamics, though it is typically a strictly structured interaction when done between two humans. However, the initial problem by this interaction in a virtual world is achieving the attention of the other party. In most games this is done by facing the avatar and being within the near proximity of the avatar. At this point there is a selection. However, this trivializes one issue. The avatar is programmed to have its own goals and, therefore, independent movement. In the case that the avatar is moving this selection becomes difficult with traditional methods. To alleviate this, many games incorporate into the avatar dynamics that the proximity of the user causes the avatar to pause its work and turn to the user and wait for the user to initiate the interaction. This is also a Dynamic Interaction, as the presence of the user changes the dynamics of the avatar.

Assuming the user can initiate a conversation with the avatar, the conversation is also composed of more interactive dynamics. The conversation in itself is an interactive dynamic, but also each component. However, here the standards of human social interaction lighten the task. There is a convention, of turn taking by a conversation

that makes the interaction of the dynamics strictly structured. When one partner of the conversation is speaking (a dynamic), by convention, the other should not interrupt. In systems that do not allow speech input this is strictly controlled through the interface. Typically, speech based interfaces either ignore the user's speech during the speech of the other, or it simply buffers it and delivers "the message" to the other after the end of their speech part.

Air Traffic Control

A final example is a project that investigated using VR to replace traditional air traffic control interfaces [KSO04]. The motivation of this project was that the traditional method of control involved the use of a 2D interface (the radar screen), which is insufficient for control of an innately 3D problem. What makes this project interesting for us is that the underlying problem is dealing with an Interactive Dynamic, but one of indirect control. Compounding this is the fact that it is a case where the problem is "hard real-time." Time cannot be stopped or slowed for interaction, a possibility in almost all other examples.

The physical planes flying around form the dynamic. Interaction with the planes happens by speaking with the planes over communication channels and instructing them what to do. The control problem becomes very difficult as the number of planes in the space increases. The traditional display only presents the height information for the plane in text form, when a plane is selected. This means the controller must keep all information in their heads and often recheck this information to keep it current. The interaction with the 3D VE representation of the world requires visually tracking planes to assure that no planes collide. Additionally, information critical to the situation must be made available to the controller, such as fuel levels, call signs, etc. To avoid information overload in the display, these aspects are only presented through selection of the spatial dynamic object. Often this is complicated by the planes being close together or even "stacked" on top of each other, e.g. with 1000 meter between them in the height field. The actions are extremely time critical and the interactions need to be easy and quick to use.

5.2.2 Design Space of Interactive Dynamics

The examples presented in the previous sub-section provide a feel for what Dynamic Interactions can be and for those that have been created in special cases. In this subsection, the design space of Interactive Dynamics is formally explored. It should be noted, however, that we consider this a preliminary look at this emerging space. We expect will expand in the coming years as more attention is given to it. The currently known design space is shown in Figure 5.4.

As can be seen in the figure, the number of Interactive Dynamics that are known is quite small. This is presumably due to the limited attention this type of interaction has had. It is hoped that this is only the beginning of future development in this way.

Starting and stopping dynamics, as an interaction, is commonplace when interaction with dynamics is performed at all. This method is the likely the most common because it is uncertain how to do anything else. Anything that changes the dynamic introduces a dynamic interaction to the dynamic, which is something that is hard to define. Section 5.3 is dedicated to looking into this issue, but it will be a recurring theme here also. The implementation of starting/stopping dynamics is straight forward, whereby any other implementation is at best difficult. However, even this simplest case depends on the next point, selecting the object of interaction.

The selection task is a common denominator of VEs usage and was defined as a Dynamic Interaction in the last section. Selection of a Dynamic is a problem that is not yet been handled. It is a difficult problem, because it is now a Dynamic Interaction and Interactive Dynamic combined, both of which have been identified as being difficult. Dynamic interaction with Dynamics will be the topic of the last section of this Chapter.

Along with selection, the idea of extending direct manipulation to Interactive Dynamics is a logical step. Unlike selection, even the approach to take in actually doing direct manipulation of Interactive Dynamics is elusive. Again, this is a case of dynamic interaction with dynamics and will be further delegated to that portion.

Viewpoint manipulation is the one of the cornerstones of VE applications, allowing the user to move through the world. In most cases it is a Dynamic Interaction as explained in the previous section. However, there are a few developments that have explored mixed systems that are Interactive Dynamics also. These systems share expand upon fixed path "canned demos" that tour the user around the VE in a fully automatic way (following some predefined path). In these systems, the user can influence the path, at least to some extent, when desired. As a prime example of an existing Dynamic Interaction with an Interactive Dynamic, a description of one of the systems is given in the next section.

All forces related haptics are interaction dynamics. The forces are dynamics, as they are exerted over time, but the user has to push back. In this way the user is also interacting. This is an interesting case, as these are in a sense a passive interaction; the user has little choice in the matter. The probably also wouldn't think if it as an interaction, except maybe when they chose not to push against the force exerted.

Entity interactions have been covered to some extent in the Dynamics chapter. For interactions with virtual characters, the most common Dynamic Interaction is conversation, a topic that has been handled previously. Probably the most wished for interaction with virtual characters is the hand shake, a Dynamic Interaction with a Dynamic. However, in this case feedback is typically the limiting factor instead of implementation details. More subtle entity interactions deal with relative positioning of the entity to some other point, for instance the user. In conversation, virtual characters need to maintain social spaces. In the bees example we saw two forms of this, grouping behavior and reactions to the user. Grouping behavior is a common portion of simple beings, but also something of interest for crowd simulation of virtual characters.

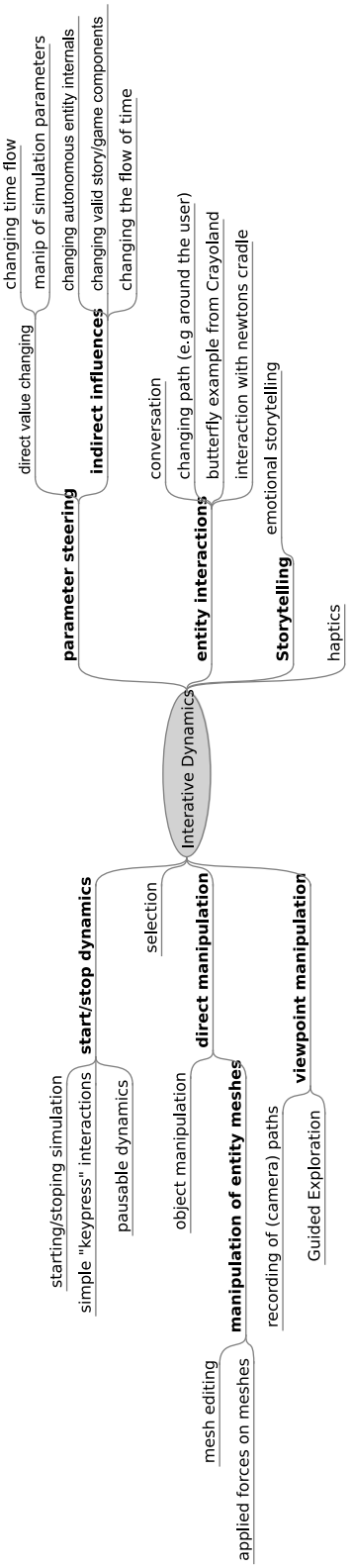


Figure 5.4: The design space of Interactive Dynamics is displayed in the form of a taxonomy.

Interaction with a simple entity can likewise take place. The basic method of discrete interactions has already been covered. Anything that is not a discrete interaction with the dynamic simple entity must again be a Dynamic Interaction with a Dynamic. An example of this will be provided in the examples chapter, Chapter 9. There, interaction with a Newton's cradle that is in motion is developed and shown.

Finally, there is the issue of parameter steering. In these cases, the user is to change parameters of the system. The first difficulty is naturally that this is typically not a visible part of the world and a fitting interaction technique has to be used. Two main categories of interactions can be identified. One is directly changing the value, such as changing the flow of time. In this case, interfaces such 2D menus can be used as commonly done in VR or specialized interface like dials or sliders. Alternatively, the user could indirectly influence the dynamics. In this case, the user changes a parameter that has an effect on the dynamics calculation at some point. An example of such interaction is performed with the sliders and knobs of the VR Tuner for adjusting an interactive storytelling application [BLM+04].

5.2.3 Implications

Interaction with Dynamics is an area that has a high potential in terms of creating interesting and exciting environments. The examples provide demonstrate a bit of that potential. However, in various places throughout the discussion, the lack of understanding of the area at the current time becomes apparent. In general the literature of VR and related fields has almost completely ignored Dynamic Interactions. However, this does not mean that Interactive Dynamics do not exist in practice, only that little formal research on the topic has been performed. We feel that the time to start this important direction is upon the community, whereby the Web3D community [BSTH01, DR03, MH06] and gaming communities (for instance the new wave of interaction devices like the Nintendo Wii remote) seem to be awakening to this currently.

When looking at the identified Interactive Dynamics, a major point comes to light: the majority of the Interaction Dynamics that are conceivable involve Dynamic Interactions. Discrete interactions with Interactive Dynamics are relatively simple to comprehend and usually to implement. However, they are in most cases not very acceptable for the user, as they don't really do what the user wants. For instance, the typical "click to start/stop" a dynamic is universally understood as a provisional interaction. In contrast, the Dynamic Interactions that are conceptually more appropriate are not straightforward. This is so much the case, that they are addressed in a separate section, which follows this one.

An assumption can be made that the reason the community has yet to fully invest in this direction is on grounds that these interactions are more difficult than the static problems that have been investigated. The prevalent thought, though unspoken, seems to be to first attempt to master interaction with static objects, before adding the complication of dynamics. The implementation methods that have been developed and used for related ideas will be discussed in the next section.

5.2.4 Requirements

The area of Interactive Dynamics remains a largely unknown space. Much of the design space turns out to fall into a further specialized category, Dynamic Interaction with Interactive Dynamics. A section dedicated to these interactions follows. Here, the system requirements that can be distilled from the discussion and observations above are presented.

Of those interactions that are not relegated to the next section, most reduces down to using an "on/off" mechanism. The various ways that this can be used and conceptualized lead to the requirements drawn. The requirements identified for this style of interaction are:

- existence of events
  - generated by external inputs (e.g. button presses)
  - generated internally (e.g. collision of hand and bee hive)
- reaction to events by
  - stopping dynamics
  - starting dynamics
  - freezing dynamics
  - continuing frozen dynamics - no time passed
  - continuing frozen dynamics - as if they had run

Having an event (discrete occurrence) type is relatively trivial, but should be an integral part of the system, allowing both internally and externally generated events. Stopping of dynamics through the interaction is a standard method that should be supported. Along with that interaction, several derived interactions are included for completeness. Starting dynamics is the complement of stopping them. Freezing dynamics is, in a sense stopping them, but implies that they can be "thawed" and run further. This contrasts to starting anew, which implies a new dynamic. When thawing the frozen dynamic two methods are conceivably desirable. They differ in the passing of time while they are frozen. In the first, no time passes for the dynamic while it is frozen. In the second case, it reacts as if the time has passed.

5.3 Dynamic Interaction with Dynamics

The investigation of Interactive Dynamics in the previous section showed that the majority of the possible interactions were Dynamic Interactions. As discussed there, such interactions are complicated, both in understanding what should be done conceptually and in implementation. This section is especially dedicated to these interactions. The interactions considered here are defined of as:

> Dynamic Interaction with Dynamics is a special subset of Interactive Dynamic that is concerned with cases when the interaction performed is in itself Dynamic in nature. Hence, the object of interaction is changing over time as well as the interaction with that object.

The difficult of this area stems primarily from the question of how the two dynamics, that of the object of interaction and the interaction, are combined.

Dynamic Interactions with Dynamics (DIDs) is a largely unexplored area and not yet formally recognized in the community. In this subsection, a basic understanding of the design space and the issues surrounding their conceptualization and implementation are developed. The following sub-section a few illustrative examples are introduced that work well on both conceptual and implementation levels. A few potential implementation methods that exist in related areas are discussed in Section 5.3.2. The discussion of DIDs in Section 5.3.3 highlights the areas for potential development and where further consideration is required. The development of requirements for a system presented in Section 5.3.4 draw on the observations of the section to create a platform of support for exploration of this interesting, yet uncertain, area of future development.

5.3.1 Illustrative Examples

Two examples of what we would consider true dynamic interactions with dynamics are presented in this subsection. The number of existing examples of this type of interaction is very limited, though individual cases can be found. The examples here have been selected to present, because they integrate the dynamic interaction into the dynamics in a smooth and integrated way that makes sense conceptually. The first example is one showing the integration of dynamic interaction with high-level dynamics. The Façade interactive drama is used here to highlight the possibilities from the realm of Interactive Storytelling. It also demonstrates a conversation base DID that is more advanced than the classic turn based interactions. The second example presents guided exploration systems, a kind of automated tour that is extended to allow the user at least some interaction capabilities.

Interactive Storytelling - Façade

The Façade interactive drama is the most widespread of the contemporary interactive storytelling environments [DMH+07, Mat02, MS03]. It is built on a custom system of the same name by Mataes and Stern. The drama plays out in an apartment with three people. The user plays one of the “actors” and two non-player actors are controlled by the system. The drama plays from a first person perspective, and the user is free to move about. Façade is a designed as a desktop experience, though an immersive, one-to-one scaled AR version was developed and tested [DMH+07]. Beyond the navigation, the only interaction available to the user is talking to the two computer controlled

actors, a couple. The plot centers on marital issues the couple, long time friends of the player, are having.

The interesting part, both for the player and the reader of this work, is the interaction that takes place. The actions that the player takes impact the flow of the story and inevitably the relationship of the couple. Although the speech recognition of the system is admittedly crude, it contains elements rarely found elsewhere. The AI system does not truly recognize the complete speech, but rather is context sensitive, inferring notions of positive and negative statements as well as about whom. It is sensitive to external environmental factors like a ringing telephone or comments about particular objects.

Where Façade differentiates itself even further is that the conversation is freeform and in near real-time. In the AR version, the conversation is in real-time. The desktop version requires the user to type their speech, as task that can be slow for many users. Typing time is exaggerated, giving the slow typist an opportunity to play. However, as with real conversations, in the Façade drama pauses in the conversation lead to discomfort for the other characters. In this way, even the inaction of the user impacts the story.

The story itself is also a DID. The underlying mechanism bases on collection of story primitives, called beats. At any moment a number of primitives are part of a possible set. Completion of the primitives leads to selection of the next, designed to build a dramatic arc. The set of possible selection is dependent on the interaction up to that point. The main mechanism used in the Façade drama is an estimation of the user's perception of the characters. One of the main goals of their system was to create an experience that can be played multiple times, while remaining unique.

Guided Exploration

Guided Exploration is an area that deals with assisting the user with the task of exploring a new environment. It is of interest in our context because it involves both an automated dynamic and interactive dynamic produced by the user. The concept is to provide at tool that is much like a guided tour, one is drawn along to all the interesting points in the environment, yet with some capability to satisfy one's on curiosity. Traditional tours of VEs are created by key-framed animation; this means that the user's position is strictly controlled by the animation. As a centerpiece of VR is the user being free to move about the world themselves, such a method is sub-optimal. Guided exploration system address this by allowing the user to have at least some influence on where they go. Two different solutions are presented here that provide methods that neatly create DIDs.

Galyean presents a method that follows "a river analogy" [Gal95]. The user is pulled along the path of the river to be shown the environment. However, in that system the user could also go in directions that were of interest to them. The "boat" continues along the path during the interaction. The user is tied to this continuously moving

point with a sort of rubber band (spring/tether in Galyean's terminology). After the maximal length of the rubber band is reached, the user is unable to deviate from the path of the boat further. This means they are drawn towards the path of the boat. This compromise is one portion the next example improves on. When looking at this strictly, the dynamics of the path following do not change, but through the Dynamic Interaction of the user, the Dynamics of the viewpoint are changed.

The Guided Exploration system from Beckhaus provides a particularly eloquent solution [Bec03]. Her system allows the user to take over control at any moment. The user is free to explore manually as long as they wish. However, when the user becomes inactive, the system restarts and begins to pull the user along to the nearest point of interest. A major difference with the Guided Exploration system is that there is no predetermined path. The system only defines the Points of Interest and in an advanced set, the best viewpoint. By moving around the user can change what the next most interesting point is when the system starts; the system them seeks out that one. While viewing a Point of Interest, whether by user or system control, the interest level of that point is reduced. When points are no longer considered interesting, the next most interesting point is made the new target. In contrast to the boat path system, the dynamics are basically turned off for the duration of the travel dynamics of the user interaction.

5.3.2 Implementing Dynamic Interaction with Dynamics

The implementation of Dynamic Interactions with Dynamics is difficult due to the requirement of merging two potentially disparate dynamics in a meaningful way. Generally, it seems that application developers are left to their own wits to create DIDs. The implementations used in the examples of the previous section are all very specialized to their applications. Only in two areas have significantly sophisticated yet potentially general methods been explored: interactions with meshes and avatars interacting with their surroundings. Methods from these areas that are potential for general application are: transitioning and morphing. Both methods are part of computer animation techniques. The methods will be introduced below with respect to the needs of this work. Animation books, such as [JNW06, Par02], provide further insight into the various methods.

In the examples presented, a major simplifying factor is inherit to the problems they address, the "object" of interaction is abstract and, hence, always implicitly active when the interaction mode is active. In the case of Façade, the story is always actively addressed with speech input. In the Guided Exploration cases, when the user provides input to the travel system, that modus is also active. For interactions that affect a specific object of the scene, a selection must be made. Unfortunately, this is also a DID and there are no significant works that address this issue. The standard selection methods are possible to use, but how viable they are in such situations is not yet known. Initial work stemming from this observation is presented in Section 10.3.2.

Transitioning

The animation of entities like avatars is typically done through a combination of methods. The basis of this is different animation methods, as described in Section 3.3. At the heart of all the methods, excepting forward kinematics, is a timed, predefined dynamic. For the sake of simplicity, the discussion here will assume a keyframed animation, meaning the position/orientation is defined at steps along the dynamic and points in-between are interpolated. Such animations are strung together to form the complete movement of the entity.

The dynamic entity is built up of a number of these primitive animations. The AI of virtual characters determines what the next dynamic is and plays the corresponding animation. The easiest to conceptualize and most often frequently used instance of this is with an avatar and a "walk cycle." The walking of an avatar is performed by looping a single gate animation. For animations this works perfectly, as the moment of change from animation to animation can be strictly controlled. However, allow user interaction means that animations can be interrupted at any moment. Take for instance that they user wants to stop walking. Stopping at the moment of interaction can lead to awkward poses of the character. This can be seen illustrated in Figure 5.5. The traditional solution, and possibly the only valid solution, is to have the walk cycle proceed to the end. This causes latency in the activity. As this is typically performed in games with a 3rd person viewpoint, this is annoying, but does not generally lead to disorientation. Non-Player Characters (NPCs) exhibiting this behavior generally seem less plausible.

Implementing a dynamic such that it stops at an appropriate moment, instead of the arbitrary moment of interaction, requires a number of things. The foremost is knowing when the appropriate moments are. This can conceivably be done in one of two ways, a priori knowledge or constraining of the system to when it can occur. This is relatively straight forward in the case of the walk cycle, but requires animator expertise. For procedural (e.g. forward kinematics) and physics based systems, this requirement is often more complicated. The system has to be programmed explicitly to handle these cases. The easiest conceivable method is to include constraints on what static poses are allowed for the end of the animation.

Knowing or calculating when to actually stop the dynamics, so that the system stops on an appropriate pose, requires a system that supports this. This makes the programming task more difficult. On the event that should stop the dynamic, the system must set some state that eventually brings the dynamic to an end. With a kinematic or keyframed walk cycle, this is simply rejecting the next loop of the animation. However, for a dynamic system this is more complex. A constraint based system requires that the "stop" constraint be coded and then handling for when the constraint is met must take the appropriate action.

Unfortunately, even this is a simplified look at the process. Typically, a walk cycle does not involve a return to a static pose, but instead is optimized to look like continuous walking (after all one doesn't return to standing during each stride in reality).

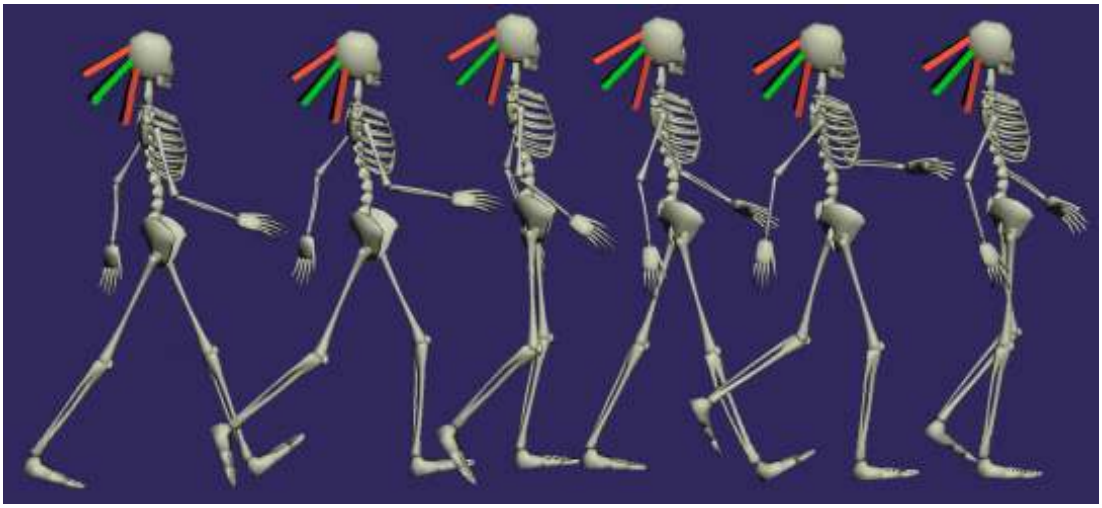


Figure 5.5: Diagram of a walk cycle. The model used is a standard part of the Cal3D animation library [cal] and is animated using it.

Instead, some method must get the character from the walk animation to the pose. One possible way is just to have an animation that goes from an endpoint of the walk animation to the standing pose. The correct transition animation need simply be queued up after the current animation, e.g. in place of the next loop of the walk cycle. An alternative solution has been developed by using a morphing technique, which will be described below. For forward-dynamics systems this equates to introducing a new set of equations that move the animation appropriate in much the same manner.

When moving to a change in the dynamics instead of to a simple pose like standing, the difficulties are increased. For instance, even the simple act of changing from a walk to a run can be difficult. Even having the character change its trajectory (i.e. turning) while walking is difficult in cases where it must look correct (unfortunately, the human is incredible sensitive to such things in humanoid movement). Typically the character just pivots without actually changing the animation. The method of building transitions between every defined animation can and has been done, but is severally limited due to the explosion of required animations. A potential solution to this problem is morphing.

Morphing

The transitions methods described above require an expert animator to create many transition sequences that are matched up to each portion. This is expensive and changes to any animation can cause many transition sequences to have to be redeveloped. Morphing is a computer supported method that attempts to overcome these limitations by automating the process. The transition is handled by different techniques, which

combine the basic animations/poses to achieve a similar effect to the hand animated transitions.

Morphing consists of a number of different techniques. The most popular methods are blending methods. In these methods, a transition is still performed, but the transition is performed via a weighted blending of two animations over time. In the context of a transition from one animation to another (or a pose), at the moment of interaction the transition starts; the two animations are combined such that the value of the original animation contributes the most to the value. At the end of the transition period, the value output is simple the new animation's value. In the time between, a function changes the weighting of the two animations that is used to determine the total value. In the simplest case, this is just a linear transition of the weighting from one animation to another. While the results of linear interpretation are not correct, they are often "good enough," and it is computationally inexpensive. Alternately, the blending can be applied to a higher-order parameter, e.g. to the velocity of each component. This then produces better results. When the animation implementation uses forward-dynamics this is particularly easy to incorporate as the animation is calculated using the higher-order variables.

More complicated weighting functions can be performed. Generally non-linear functions give more visually acceptable results. Blending with sinus functions produces acceptable results with minimal additional computation costs. See an animation book for more details to implementing these methods. Other algorithms perform higher-order blending, particularly at the end points of the transition. These methods address issues of discontinuities in the motion at the end points. When performed externally to the implementation, for instance as required when using keyframing, extra functions handle the transition of the transition. These functions are commonly referred to as *in/out functions* or ease-in and ease-out functions. 2^{nd} order algorithms for smoothing the final moments of the transitions are fairly common in advanced systems when the perception requires realistic movements. Viewpoint control uses such methods in almost all cases, as the transitions can be very disturbing otherwise. Most often this is manifest in camera control algorithms for games.

Morphing is not only an important area for animation transitions, but also for interaction with surfaces. Naturally, this also applies to interaction with dynamic surfaces. Such interactions can be envisioned for applications within haptics. The implementation involved with the morphing portion is similar, though more complicated than what we have been talking about previously. Surfaces in the sense meant here are connected meshes. When a single point is deformed, the points adjacent to it are also deformed. This is a large area of research in itself and quite complicated. Perhaps one of the most interesting starting points for our work is that of skinning. Skinning is a process of providing deformation for avatars when moved. Beyond it being fairly common and also efficient, it uses a series of constraints similar to the method used by Galyean, though with more constraints at the same time. Animation books in general give some insight into this area, and books, such as [GDCV99], focus heavily on the ideas behind this. The morphing of surfaces is beyond the scope of this work, though its methods may proof useful for future developments.

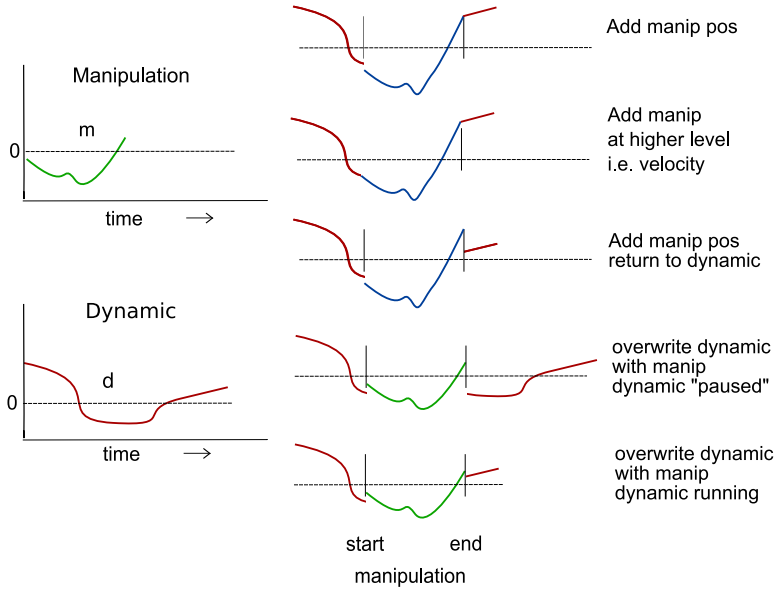


Figure 5.6: This diagram shows how the identified methods in which direct manipulation and existing dynamic can be combined. For simplicity, the interaction is reduced to a 2D space. The timings of the interaction and the dynamic are seen on the left hand side.

5.3.3 Discussion

This section has introduced Dynamic Interactions with Dynamics as a formal area of interest via examples and some possible implementation methods. Throughout this introduction a number of points have been touched on that should be addressed in more detail. Well functioning examples of DIDs have been presented, but a general concept to how DIDs should work in principle is missing. A discussion of what makes DIDs so conceptually difficult highlights this fact. Partially because of the uncertainty to how DIDs should be performed, defining how to implement these interactions and dynamics is difficult. A few more general support mechanisms were research and presented in the previous sub-section. A discussion of the implementation issues is provided. In the final sub-section below, the system support mechanisms that are likely to meet developer needs, at least for exploration of the space, are presented.

What exactly should a Dynamic Interaction with a Dynamic entail? The examples shown were simple enough to grasp relatively easily. However, when trying to extrapolate from those examples to come to an "essence" of DIDs, one quickly runs into difficulties. Likewise, defining DIDs is simple enough, but does not really help in understanding how they should work. To illustrate this, an example will be made of classic direct manipulation in VR. This interaction is a good example because it is both easy to visualize, but also something that is commonly thought of (though according to the author's experience, always quickly disregarded). Assuming the manipulation

being performed is translation, looking at the possible ways in which the dynamic and the manipulation Dynamic can be combined is informative. Figure 5.6 visuals the possibilities:

- add the manipulation to the continuing object dynamics - changing the position directly
- add the manipulation to the continuing object dynamics - changing the dynamic at a higher-order level (e.g. velocity)
- add the manipulation to the continuing object dynamics - snap back to dynamics afterwards
- stop the dynamics and manipulate - continue as before (i.e. the manipulation has no affect after manipulation ends)
- stop the dynamics and manipulate - continue dynamics from the manipulated position

Implementations for each of these possibilities are easily created; the larger question is, what manipulations make sense? We feel that the answer to that lies in the application for which it is needed, making it very difficult to answer question. For that reason, DIDs will not be heavily explored in this dissertation, but rather left to future work. We feel this is justified, as the foundational experience with dynamic interactions and even interactive dynamics are not yet present. This dissertation begins to address those issues and hopefully lays the foundations for future work in that direction. However, supporting the exploration of the area with our current understanding is desirable.

Highlighted throughout the presentation is the need for programming the dynamics aware of the interaction. For instance, only one of the dynamic manipulation possibilities above works without a method that is aware of the interaction (and it is debatable whether it should even count as an interaction as it doesn't actually have an effect on the Dynamic after the interaction ends). Even then a minimal awareness is assumed, the ability to stop the Dynamic and restart it. In restarting it, we have to consider that time has continued, in the real world and potentially for other portions of VE also. Either time continues after the interaction as no time is passed, or time passes for the Dynamic the whole time and the object "jumps" to the correct position is if it has moved via the dynamics the entire time. The other possible methods imply that the implementation can handle the interaction itself or at least the result of the interaction.

Certain programming approaches may prove advantageous for DIDs. The traditional approach in the community is to use Object Oriented programming methods. Since the dynamic has to be aware for all but the trivial cases, this implies that the objects have to be extended to include this awareness. This has the potential to cause an explosion of classes, as every different interaction and dynamic combination requires unique implementation.

Functional programming and constraint based methods allow more generic and simpler inclusion of the interaction in the implementations. In a pure functional programming environment the functions receive input every execution and change it into

a new output. Since the function only acts upon the input, interaction that changes this input does not change the dynamic functions ability to continue working. However, approaches such as changing higher-order functionality have to be merged into the implementations themselves, potentially leading to the same explosion issues. In constraint based systems, extra constraints relating to an interaction channel could be established. The system need only to have the ability to switch on constraints at run-time so that the interaction is only present when desired. Several of the existing systems allow such functionality, see Section 6.2.

In order to be able to perform any interaction in a VE, the computer first has to be aware of the "object" of interaction. This is a process called selection that is itself a dynamic. As such, before any other interactions of this class can be explored, methods to select dynamic objects are required. The traditional methods of selection, described in Section 6.1.1, should function for this task. However, if one considers that many sports are based on the challenges that interacting with dynamic object entail, methods that already have known shortcomings (for instance ray selection of small objects) can be expect to perform poorly. Because this is so critical, initial work investigating methods of selection of dynamic objects has been performed as a derivative work of this dissertation. The initial results of that study are introduced in Section 10.3.2.

5.3.4 Requirements

As remarked in the discussion, general guideline how DIDs should appear cannot yet be created. Likewise, drawing requirements as done for the other areas is difficult. As DIDs are a specialization of Dynamic Interactions the few general requirements that can be developed overlap with those. For instance continuous, interval, and set time representations need to be supported. However, it makes sense to introduce requirements that supporting the identified implementation methods, i.e. supporting handling of transitions.

Fundamental to the transition idea is that there is an event that indicates when to change. Events are already included in the previous sections as a fundamental element that is required. The types of transitioning functions that should be supported are:

- in/out functions,
- blending,
- user defined transition functions.

To support those transitions, a few further requirements are placed on the developments to facilitate the exploration of the design space:

- exchangeable (interpolation) kernels,
- basic sets of interpolation kernels (linear, sinus, etc),
- support for different timings of the functions.

Part II

System Development and Implementation

In this part of the dissertation, a system of support for the creation of the type of environments identified as Dynamic, Interactive Virtual Environments is developed. The developed system focuses specifically on the creation of DIVEs for usage in a Virtual Reality context. As such, the context of Virtual Reality is presented in the first chapter of this part, Chapter 6. The Virtual Reality chapter also presents and discusses the existing relevant support for DIVE like functionalities. Chapter 7 develops a system of support and introduces the implementation of Functional Reactive Virtual Reality (FRVR). The system developed is built on a programming paradigm, Functional Reactive Programming. An introduction, for those not familiar with it, is found in Appendix A.

Chapter 8 presents extensions to the basic system of support developed. These extensions complete the functionality of FRVR, such that it fulfills the requirements developed in Part I. This chapter technical in nature and may require reading the Functional Reactive Programming Appendix to full understand it. Finally, Chapter 9 shows how FRVR can be used to create DIVEs. The examples of that chapter demonstrate how use cases previously developed can be performed in FRVR. Chapter 10 concludes the dissertation and identifies directions for further research and work.

Chapter 6

Virtual Reality and Existing DIVE Support

The first half of this dissertation has focused on the creation of Virtual Environments that meet certain standards, specifically that contain dynamics and interaction. The analysis of such environments was largely held general, though interactivity and real-time capabilities were implied. In this second half, the development of supporting mechanisms will be created in the context of Virtual Reality. The choice of Virtual Reality as a backdrop for the developments is in one sense arbitrary based on the authors background, but this choice also serves a purpose. Virtual Reality is the most technically demanding of the various areas that make use of VEs, so the results of this work are applicable across the spectrum of VE usage.

This chapter is divided into two sections. In the first half, the field of Virtual Reality is presented. The introduction will be kept short and cover only the relevant materials to this dissertation. The second half presents a survey of existing software systems that are relevant to the goals of the dissertation. This will primarily focus on VR systems, though a few particularly relevant specialized systems are also described.

6.1 Virtual Reality

This section serves as an introduction into the field of Virtual Reality. Numerous books cover the field of Virtual Reality [BC03, BKLP05, Rhe91, SC03] in depth, so this introduction will focus primarily on the components that are important to the dissertation. This includes understanding the basic concepts of Virtual Reality as well as certain details that influence the implementation of the system to develop. In the first part, a high level explanation of the field is provided. Building from that description, the final two sections present necessary background on Virtual Reality hardware and software respectively.

6.1.1 Basics

Virtual Reality is a term that has been used to mean many different things. This part serves to clarify what Virtual Reality (VR) is and to provide the reader with an understanding of what makes VR unique in the areas of VE usage. Because of the confusion over the field, a definition for Virtual Reality is given to provide a firm understanding of the area. The technical details of VR are relegated to the following parts.

We define Virtual Reality as:

> Virtual Reality is the presentation of an interactive, synthetic, computer generated, spatial environment, which may or may not reflect the "real world," in real-time in an immersive display.

This definition is highly influenced by Carolina Cruz-Neira, though is an unpublished description from her. That a Virtual Environment is the center of the VR experience is easily identifiable within the definition through the similarities in the definitions. Three important additional qualifiers are added in this definition. Interactivity is a hallmark of Virtual Reality. Interactivity is a central theme of this dissertation and will be explored more below. The additional requirement of a real-time presentation is extremely important for VR. This is important on various levels in VR, ranging from being required for effective interaction to ensuring that the user does not become sick. The final point is the presentation of the VE in an immersive display.

The real-time and interactive qualifiers reduce the possible systems to those of a few areas; those can roughly be identified as computer games and VR. Immersion is the final quantifier that separates Virtual Reality from computer games[1]. Although this term is a key to the definition provided, it is not a particularly easy term to define. One of the issues is that immersion is an abstract quantity. The other issue is that there is significant overlap of its meaning with that of another term, presence, making it even more difficult to concretize. The real-time aspects of VR are discussed in the interaction sub-section that follows as well as in the technical discussion that follows. Presence and Immersion will be discussed in more depth in the second sub-section.

Interaction

The area of interaction is a fairly well defined area in general, but in the case of Virtual Reality its understanding will be called somewhat into question in this dissertation. Before presenting the standard VR understanding of interaction a definition will be put forth. This is done to clarify our understanding of the term and to provide a definition in a space that is not normally formalized.

[1] It should be noted that the line sperating VR and games continues to deteriorate, as VR like interfaces become more mainstream.

We define interaction as:

> The user taking influence on the environment or its controlling structure

This definition is purposefully left extremely broad. The definition explicitly refrains from any mention of either the means of interaction or its semantics. The definition also explicitly refrains from phrasing interaction in terms of interaction with an object. Such phrasing implicitly implies that interaction takes place only in the typical direct interaction form. As can be seen in Chapter 5, direct interaction is only one possibility. Instead of focusing on the object of interaction, the definition squarely focuses on the user.

Perhaps the most radical portion of this definition is that the user is the focus of the statement. The definition takes cues from two places. The definition most closely resembles the discussion of interaction by Laurel in the development of her agency ideas, see the immersion and presence sub-section that follows or [Lau93]. The phrasing "the user taking influence" is inspired by that discussion. However, the interpretation of this statement takes its cues from a new perspective on interaction in VR and related areas, pushed by Beckhaus and Kruijff [BK04]. They approached their discussion of interaction from the perspective that the user is the center of interaction. In the setting of Virtual Reality, user centered computing takes on a new meaning, as the user is literally in the center of the interface.

VR is a medium of computer mediated interactions. As with all computer human interaction, it can be simplified into exchanges between a human (or more humans) and the computer. This exchange is predominately represented as a cycle, where the human's input affects the output of the display. The loop is completed through the displayed output that the user views. Figure 6.1 is a representation of this cycle.

Depending on the observer's point of view when looking at Figure 6.1, the input comes from the human and output out of the computer, or the input for the human comes from the computer and the computer detects the human's output. Although we support the viewpoint that the human should be the focal point, this dissertation follows the system denotation. This is done, since the system view fits with the standard language usage in VR system writings and the basic books in VR. In the follow paragraphs the classical VR view will be presented. However, much of the motivation is in being better able to create VEs that the user responds to. This viewpoint is important in several of the ideas that come out of this research.

Interaction is a very active area of research in Virtual Reality, with a recent book dedicated to the topic [BKLP05]. Even so, interaction in VR remains a somewhat elusive area. There are not yet any standards in place for interaction nor interaction devices. Interaction research tends to be rather narrow in scope, usually focusing on a new prototype device or prototype method. An introduction to the most common classes of devices in VR is provided in Section 6.1.2. The pseudo standard view of interaction in VR is Bowman's interaction taxonomy [BH99, BKLP05, Bow99].

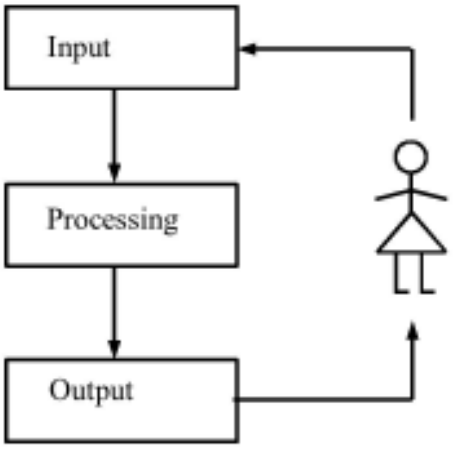


Figure 6.1: The interaction loop labeled from a system viewpoint.

At the highest level of the taxonomy there are four basic categories of interaction:

- Navigation
- Selection
- Manipulation
- System

Navigation is the most common interaction in VR and deals with how one moves throughout the VE. Although termed navigation in Bowman's works, the description provided can more accurately be termed *travel*. Other works differentiate between wayfinding - roughly the cognitive processing leading one to know what action to take - and travel - the action movement through the environment. Selection and manipulation are the next most common interactions. Manipulation is dependent on selection in VR, as the computer needs to know what "object" should be interacted with. Manipulation typically reduces to spatial movement, as discussed in Section 5.1. Finally the system category is something of a "catch all" category, used for containing interactions that do not fit into the other categories. The main component is typically symbolic input, e.g. text input and general system control.

Figure 6.2 shows the most time relevant portions of the selection and manipulation branches of the taxonomy. Selection is the process by which the user indicates an object of interest in the environment and initializes another activity, namely manipulation. Selection is typically performed using a hand-held 3D pointing device. Since most methods are designed to allow selection of objects beyond hand reach, the process is often two-fold. First the user must meet a pre-selection of the object, before indicating the use of a manipulation method, for example with a button press. Visual feedback is a crucial component of this process. Without such a method, the user is often uncertain

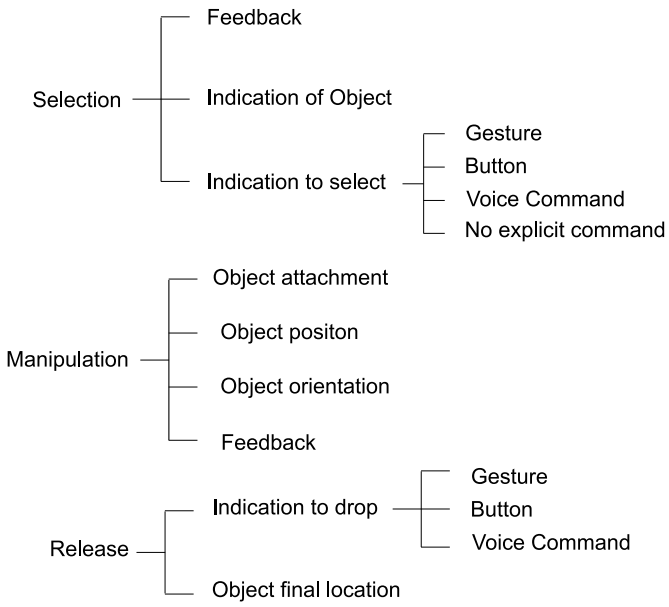


Figure 6.2: Selected portions of Bowman's taxonomy for Selection, Manipulation, and Release of objects in VR [BH99]. Only the most relevant portions of the full tree are shown here.

of the object that would be selected if they press the button. Coloring the object that is currently pointed to by the selection method is a typical feedback method, a form of pre-selection.

Manipulations considered in the taxonomy are an array of "world design" type interactions. Translating and rotating objects are the most common of these. However, many other interactions can exist. In Chapter 5 the discussion of interactions pointed out that spatial manipulation is just a small portion of the possible interactions. The feedback of a selection method actually performs a manipulation of the object, changing its color. Some application spaces involve investigating and changing non-visible aspects of objects. Examples of this can be found in many science and engineering applications, where the identities of specific objects are needed as well as other parameters that control an underlying simulation.

The final area of interaction is that of system interactions. As with the manipulations of data above, this category deals with manipulation of external data. Typical interactions involve data entry and interaction with the underlying operating system. The system category is, in general, under-defined and rarely investigated. Symbolic (textual) input is the sole area that is usually considered.

Immersion and Presence

A decisive point in separating Virtual Reality from various other related areas that are based on VEs is immersion. Although, this term is key in the definition of VR provided above, it is not a particularly easy term to define. Part of the problem is its intertwined meaning with the term presence throughout much of the literature. The best way to explain immersion is through allegory with other mediums. Samuel Taylor Coleridge is accredited with a phrase that is most commonly known in relationship to the film industry and roughly explains presence, "the willing suspension of disbelief" [Lau93]. This is the idea that the user is willing to forgo focusing on their physical surroundings and instead choose to believe they are in the presented environment. The user is said to be *present* in the VE. Immersion is surrounding the user in the presentation of the Virtual Environment such that they find themselves present in the VE.

Because presence and immersion are difficult topics to understand, the following paragraphs will introduce the ideas. The simple way to separate is to say that immersion is the physical properties that enable presence and that presence is a psychological process that allows one to believe in the VE. Unfortunately, even assuming this proposed separation can be held, the processes are still intertwined. Because the research of this dissertation has implications in these areas, the topic is further explored.

Presence was introduced by Slater et al. as a measurement of the effectiveness of the environment and VR display and is now the defacto standard in VR [SS00]. However, presence as a usability measurement is not without its issues. There is no direct way to measure presence. Instead, presence is measured through various self-report questionnaires. However, Slater et al. developed a new metric to detect how often the user is torn away from the virtual reality into the physical reality [BSS03, SS00]. Numerous factors can cause such "breaks in presence." Why this is important is that they showed that such *Breaks-in-Presence* (BIPs) can be detected by physiological factors and positively correlate directly to traditional presence measures. This creates an objective measure of presence. More complete reviews of presence research can be found in [SS02] and [IJs02].

Numerous factors affect the viewer's level of presence in the generated environment [SS02]. Although Sadowski and Stanney identify seven factors, three relevant factors for this dissertation are: "system factors," realistic environments, and agency. The system factors can broadly be explained as standard VR system requirements. For example, the system must runs at an appropriate frame rates, without significant delay, and without other configuration issues. According to Sherman and Craig [SC03], 15Hz is "marginally acceptable" and above 30Hz is considered "very good." So called "immersive displays" are one contributing factor. Immersive displays are described in Section 6.1.2. A commonality of most displays is that they surround the viewer, blocking out the physical environment around the viewer and replacing with the artificially created environment.

We have presented that immersion is mostly about the physical presentation and its characteristics, i.e. the "system factors" of Sadowski and Stanney. Whitton takes

a similar stance on the term, calling immersion "the stimuli that collectively represent the virtual world" [Whi03]. She differentiates between the "degree of immersion" and the "quality of immersion." The degree of immersion, she says, is dependent on: the number of modalities addressed and how well the user is isolated from the real world. She says that the quality of immersion varies with the "fidelity of physical simulations, rendering (for all senses), and presentation/display of the data." More interesting are the example factors that Whitton presents [Whi03]:

- geometric resolution of models,
- time resolution of a particle-system simulation of falling water,
- vehicle simulation physics,
- how the graphics simulate the physics of light transport,
- detection of and realistic response to collisions between objects,
- display field-of-view, resolution, brightness, and refresh rate,
- frequency response of speakers or headphones,
- processor speed, and
- latency of response to user input.

Bowman and McMahan further explore the effect on immersion for different physical factors of display technologies [BM07]. That paper collects much of the research on the effect of visual display technologies on immersion.

Generating realistic environments is factor to the user's presence in VR systems, or at least it is often conjectured that it does. In VR, this has traditionally created a focus on making photo-realistic environments. Topics such as the effects of shadowing on presence have been investigated [KYMS06, SMC95, WBL+96, ZP03]. The results of these studies, however, have shown that realism has no significant impact on self-report presence. Only [KYMS06] finds a positive correlation. It is interesting to note, however, that much of this work has been performed in the early days of presence research, the mid to late 1990s. The age of the research has a significant impact as the imagery used was very primitive, calling into question the legitimacy of the results in today's environments. For instance, in often cited research from Slater et al. about the effect of shadows [SMC95], the entire scene consisted of 413 triangles, with the main objects being 2 boxes! Although not studied, it can be hypothesized based on the research that significantly incorrect modelled worlds could cause breaks in presence; this could for instance, be with incorrectly modelled dynamics. This hypothesis is the basis for work done by the author resulting from this dissertation [Blo07a, Blo07b].

The final component of consequence in this dissertation is user *agency*, "the power to take action" [Lau93]. The agency factor in presence is identified by Sadowski and Stanney as "user-initiated control" [SS02], and by most simply as interaction. However, interaction tends to convey the idea of object based (direct) interaction instead in contrast to "the power to take action." A good survey of interaction and its affect presence can be found in [vdS00]. The idea of agency is important in its own right in this discussion. User Agency is a key factor in Virtual Reality, at least theoretically.

In VR, agency is achieved through interaction with the environment. If user agency is achieved, then they are likely also to be present in the environment. Unfortunately, as noted by Laurel and supported by the BIP theory, when interaction either is not present or fails the user is torn out of the immersive illusion and returns to the physical world. This forms a sort of double edged blade, where interaction is required to achieve presence, but creates a high probability of destroying presence if not done correctly.

6.1.2 Hardware

This section introduces the basic hardware requirements of Virtual Reality. The presentation divides them into four parts of similar functionality. The initial sub-section discusses the different modalities addressed by VR and how they are employed. Because the primary output channel used is the visual system, those hardware systems are first addressed in their own part. The output components of the remaining modalities are then presented. The necessary computing hardware for driving those outputs and collecting the inputs is discussed. Finally, the basic input devices that are common across the field are introduced. In addition to the general VR books mentioned previously, the "3DUI" book is recommended for the interested reader [BKLP05].

Input and Output Modalities

The main modality addressed in Virtual Environments to date is the visual channel of the users. This is presumably due to the dominance of the visual channel in humans. Input is generally performed by tracking of the body and hand coupled devices, in other words very limited forms of haptics. When another channel is incorporated it is most usually the auditory channel. Sound output is not uncommon. The auditory channel is the most frequent channel used as "unconventional" input into the system, usually speech input. Haptics, the touch and proprioceptive senses, are also areas of interest and of some use in VR. These interfaces truly address these senses as opposed to the usual hand held devices. Most of this work is performed within a specialized "haptics" community, though there is overlap. Finally, there is some work with olfactory and even taste input modalities. A good survey of all of the input and output possibilities across the modalities, including both the hardware and the human, can be found in the SIGGRAPH UCHI course notes [BK04]. The visual output of the system is the basis for majority of the VR research and is the component almost always used in VR setups. Because of this importance the visual output system will be discussed on its own in the following section.

Auditory components of VR systems generally fall into one of three categories: background music, (object) specific sounds, and sound inputs from the user. A more thorough survey of sound usage in VR can be found in Sherman and Craig [SC03]. Background music is sometimes used in VR systems in much the way that it is used in movies, to create ambiance. Occasionally, in the more artistic applications, the background sounds play an important role and are chosen to pass to the current situation

or area of the VE. In some VEs, specific objects may have sounds "attached" to them. This can be anything from the sound of the bird flying by, to a human speaking. Generally these sounds are localized and only audible in the presence of the object. A good overview of spatial audio for VEs can be found in [KJM03] and a VR explanation can be found in [SC03]. Auditory input is typically in the form of user speech. This is either in the form of communication with avatars or as command input, e.g. "put that there" types of commands.

Haptics is a complete research area in its own right. A good starting point for investigating haptics is [SCB04]. In general, haptics are used as both an input and output modality. Haptics address two of the potential senses of the user: touch or tactile and proprioception. Tactile output devices generally provide the feel of objects, though less sophisticated devices, like "pop through buttons," exist that provide very limited information [BKLP05]. Proprioceptive devices effect the movement of the user, usually their limbs. These devices work by exerting forces on the user that correspond to forces from virtual objects. Even so-called passive devices exert a force on the user, although only a resistive force. Because of this dual nature, proprioceptive haptics are almost always input and output modalities simultaneously. Addressing the haptics modalities is typically limited by the hardware solutions available, and they are applied only for specific problem spaces. The adoption of haptics has meet success, however, in the medical fields, particularly for training applications.

Research on olfactory (smell) output has been limited, but has an ever increasing group of researching investigating it [Che06, HB06, NNHY06, NOK+08]. Due to the complex nature of scents, it is technically difficult to set in place. Smell is not only complex in their creation, but also in effect they have on the user. This is because the sense of smell is processed near the emotional centers of the brain [Has06]. However, this emotional coupling makes them a potentially powerful component of VEs. The final challenge of olfactory displays is the time component. Unlike the other modalities, smells have a number of time based factors: they dissipate slowly, users adjust to the smell, etc. To effectively use them requires spaces specially designed for them. In particular, the physical space needs to enable dissipation that matches that of the virtual world. For instance, as the user moves away from an area a strong scent has to disappear quickly from the physical space. Building systems that can evacuate the smell quickly is difficult if auditory and immersive displays are co-located in that space.

Visual Output Devices

The standard VR output devices attend predominately to the visual system of the user. The visual modality is covered by a large spectrum of technologies and devices. These range from desktop VR systems up to large Immersive Projection Technologies(IPTs). In general, to meet the sensory capabilities of the human visual system, the systems attempt to cover as much of the visual field as possible of the user and blend out the real world. One commonality to most VR systems, is producing a stereoscopic image; that is, each eye sees an image calculated for its position.

Figure 6.3: A user is pictured wearing a current generation HMD, the Cybermind Visette45 SXGA.

The first dedicated VR systems and the cheapest of the systems that truly support immersion are Head Mounted Displays (HMD). A HMD is precisely as the name alludes to, a display or set of displays that is affixed to the user's head in some manner. Ideally, HMDs would like setting on a pair of goggles and certain versions of come very close to meeting this ideal. Figure 6.3 shows a current generation HMD in use.

While various HMD technologies have been developed, there are a few commonalities to them. Images have to be delivered to the eyes in some way. This is done either by placing the display directly in front of the eye or by using a set of mirrors to route the image. The most basic difference is whether the display produces one image or one image per eye, i.e. monoscopic vs. stereoscopic displays. For stereoscopic displays, HMDs have two independent displays. Two further factors are important. Naturally the resolution of the displays impacts the fidelity of the display, particularly since the displays are so close to the eye. Low resolution displays appear very grainy in this constellation. Until recently, the resolution of most displays was limited to 640x480, which severely limited the visual impression of the world. The HMD in Figure 6.3 has a $1024x768$ resolution per eye. The second factor is the Field-of-View of the display. Because the human has high acuity in this regard, a greater FoV (> 60) improves the user's immersion. Unfortunately, high FoVs remain only possible in the most expensive HMDs of the current generation. All HMDs require some form of tracking to allow the computer to receive information as to the position or at least orientation of the user in the real space, described further in Section 6.1.2. The image displayed is calculated to the viewpoint of the user.

The main concurrent to HMDs are Immersive Projective Technologies (IPTs). These displays attempt to cover the user's field of view by using large displays. These displays are driven by projection systems, hence the name. The classical IPT is the CAVE™. The CAVE™ is formed as a cube, where four sides (the front, left, right, and floor) have images projected onto them. These IPTs are generally around $3m$ by $3m$ by $3m$ in size. Another popular format of IPTs is a "wall." These are single surface projections, but are generally of higher resolution and often (much) larger sizes. Generally, CAVE-like IPTs are stereoscopic displays, where wall displays may be either mono or stereo. Stereo projections are usually produced in one of two ways: either passive stereo or active stereo. In both cases two images are projected onto the same surface, one per eye. The difference lies in how the images are separated. The two most popular approaches are polarization (separation in light space) or time-multiplexed (separation in time). Polarization is a form of passive stereo, requiring special polarized lenses in front of the eyes and projector. The time-multiplexed solution is called active stereo. It interlaces the images for left and right eye one after the other. Special glasses that are synchronized to the image update are required to view the images in separated form. These glasses shutter each eye when the image for the other eye is shown. Figure 6.4 shows an IPT at the University of Hamburg, a so called "L-shaped" IPT, which is a front and floor projection system that uses circular polarization.

As with the HMDs, there are several factors that influence the effectiveness of the display system. For immersion many of the same factors are important. The Field of View of the system is important, as well as the resolution of the displays. IPTs have other issues in their design that have to be addressed. The separation of images for the eyes can be problematic depending on the method used. The most common issue is so-called "bleed through" in which the eye sees a ghost image that belongs to the display for the other eye. This is most predominant in passive displays. The other issue is related to the requirement of synchronizing multiple displays. IPTs may be composed of up to a 100 different projectors that build a single image, though a range from 4-12 is more common. If the image updates are not synchronized this can either destroy stereopsis or just be very disturbing. This involves both the update of the application across machines and also the update of the imagery. The update of imagery is either done via synchronized swapping of the new images or, in the case of active stereo, at the level of requiring synchronization of the image generation units clocks (see the Computing Hardware part below). Complete discussions of these issues can be found in any VR book. Finally, the displays need to update at a rate that is suitable for the visual channel. Generally, this is at least 50Hz per eye and higher frequencies are desirable.

Devices of Other Output Modalities

The hardware that supports other output modalities is less well established in VR. The auditory output systems are the most standardized of this group. The hardware of the haptics and olfactory modalities generally fall into the experimental area. Complete

Figure 6.4: Here the user can be seen in the imveL, an L-shaped large scale projection based VR system. The system include 3D sound system and tracking via an optical tracking system.

coverage of the different hardware possibilities is avoided in this treatment. Instead the focus will be on the aspects of the solutions that are necessary to understand for applications that would use/steer them.

Auditory output systems come in two variations head coupled and speaker systems. Head coupled systems are often found in conjunction with HMDs or in environments where multiple users share the same physical space, but not the same audio space. The alternative approach attempts to surround the user via placement of speakers in the area of the display. The most well know difference beyond this is the number of channels supported, e.g. stereo, 5.1, or 10 channel arrays. An example setup is the surround sound system shown in Figure 6.4. There the system consists of four speakers placed at approximately head height at the four corners of the room. The setup of such systems is beyond the scope of this work. What is important is to realize that many channels may have to be controlled and how the channels are controlled can be of critical importance. In some systems truly 3D sound is produced from objects in the virtual world to the observer in the real world, i.e. spatial audio [KJM03].

This may be very involved, including the need for head tracking and even calibration to the user's unique ear structure. Because sound in composed of such high frequencies, the hardware requires updating at higher rates. Standard frequencies for digital sound signals are 44KHz.

Haptics systems are more varied and the area is still in development. Various types of hardware exist, many of them considered unconventional [BK04, SCB04]. Many haptics systems are attached to the ground and apply forces to the user. Others are attached to the body, for instance to the hand and fingers to enable grasping virtual objects. For the purposes of our work, the devices themselves are not as important as the requirements they place on the system. The largest point when producing haptics is that the refresh rates and interaction rate required for acceptable usability are much higher than those of the visual channel. 1000Hz refresh rate for the force based devices is often required and sometimes move. Tactile inductors require much lower update rates, depending on the receptors being activated 0 – 10Hz or 50 – 500Hz. A discussion of this can be found in [BC03]. The disparity between an 1000Hz (or even 300Hz) update rate required for haptics and the VR system update rate of 30 Hz is a perennial issue of integrating haptics and graphics based systems.

Computing Hardware

Virtual Reality generally has only a few truly unique requirements on the computing hardware used. The most basic of the requirements are those that apply to any modern "gamer" system. The most important factor for most VR systems is the graphic engine (card) and its connection with the remainder of the components. Where Virtual Reality system requirements break from the standard is on two fronts. The first is the requirement of more screens. For multi-screen displays this means having to produce a potentially large number of visuals simultaneously. This is compounded by the fact that all screens have to refresh at the same time. The second is that of stereo images. In the case of the IPT we have seen there are differing solutions involving this.

The simplest solution for the VR developer is unfortunately today a rarity is the use of a *compute server* with special *graphics engines*. This area was spearheaded by SGI™ corporation. The SGI™ supercomputers were built with a shared memory system, where the graphics engines (commonly with 4-6) all had access to it as local memory. The graphics engines also all shared the back end up, meaning they could all receive graphics commands simultaneously, thereby reducing bandwidth needs. These graphics engines were specially built to support "active" stereo; this was achieved by providing a method to synchronize the graphics engines at the clock level.

The modern solution, and ironically the predecessor to the compute servers, is clustered computing. In this paradigm, each computer is basically a modern "gamer" PC. Generally, the graphic cards do not support active stereo synchronization (clock level sync), though some high end cards to do have a hardware "swap lock" capability. Partially due to this limitation, the clusters typically drive passive stereo IPTs. Most modern HMDs can process the separate images of a dual head PC card. The required

frame synchronization is instead performed via software tools (software swap lock), providing an approximately same change to the new images.

This solution has all of the normal issues of clustered computing. Naturally, all of the information of the scene being rendered, the input, and the current state changes need to be present on each machine. This distributed computing is then supported through differing networking paradigms. A somewhat dated reference on distributed VR can be found in [SZ99]. Because of the requirements for synchronizing and the sometimes large amounts of data transfer, such clusters are often setup on their own isolated network.

Finally, there is the issue of input devices and other modalities. Wired input devices generally have to be positioned as close to the interaction space as possible. The graphics engines are often located near projectors, potentially located 10m or more way from the system. Because of this, a device PC is often placed near the system. Usually a low end PC can be used. Other components like tracking system generally have their own dedicated hardware components or another PC for processing data (for instance triangulation of tracking in optical tracking). As mentioned in the previous part, other modalities often require running at higher update rates, basically ensuring the need for dedicated machines to run time. For instance, most systems that support sound have a dedicated sound computer with specialized hardware. Similarly, haptics has very specific and stringent hardware requirements that restrict them to a specialized machine.

Input Devices

Input devices in VR differ from standard desktop computer devices out of necessity. The user is typically interacting in a large space with freedom of movement. The most critical of all "input" devices is the tracking system. The tracker is responsible for determining the position of the user and any interaction devices. The number of developed input devices that have been developed for VR is very large. We present here only the most common and ubiquitous devices: the "wand" and the glove. The important factors for this work are well illustrated by those examples. Interested readers should reference [BKLP05] for an overview of basic devices and [BK04] for a survey of less conventional interaction devices.

In order to calculate spatial audio or perspectively correct images for IPTs or images for a HMD the position and orientation of the user in the physical space of the VR system needs to be determined. This task is performed by a tracking system. Tracking systems are generally classified by the number of "degrees of freedom" they can determine. For trackers, degrees of freedom (DOFs) indicate determination of either position or orientation on an axis. 6DOF trackers are the standard, as they must determine both the position and orientation of the tracked devices. Trackers typically run at high refresh rates, between 60Hz and 1000Hz, though 100Hz can be expected.

(a) A wand built from a Nintendo Wii controller with an attached optical tracking target

(b) A glove input device with bend sensors and an attached electromagnetic tracking target

Figure 6.5: Typical VR Inputs Devices

Of the various types of tracking systems the most common fall into one of two major classes. The popular electro-magnetic trackers detect electro-magnetic fields in small tracking targets to determine the position and orientation. The fields are induced by special emitters that are part of the system and must be mounted into the VR systems' space. Most systems are capable of tracking multiple targets. An example of one of the tracking targets can be seen in Figure 6.5(b) as the small gray box on the wrist of the user. The other major class of trackers is line of sight trackers. Large space trackers of this type typically use optical systems. Through means of triangulation between multiple cameras (or conversely the camera is worn and multiple "targets are triangulated), the position and orientation of a target is found. For VR systems the most common usage of this is tracking of special "retro-reflective" targets and infra-red cameras. The "tree" target attached to the front of the "wand" in Figure 6.5(a) shows such a device.

The *wand* is probably the most prevalent input device in VR systems. However, exactly what a wand looks like and what interaction possibilities it has are not standardized. In general, the wand device is a handheld device that resembles (in an abstract way) a magician's wand. An example "wand" can be seen in Figure 6.5(a). It is tracked in 6D (3 positional and 3 rotation axis) and usually has a number of input buttons. The user of the VR system holds this in their hand, triggering actions with the buttons. Generally, the wand is used for all interactions.

The second most common input device is the glove. This device comes in two variations, but both are basically as expected, a glove with special sensors. Additionally, the user's wrist is often tracked in 3D space. The basic difference in gloves is the sensing

technology used. The simplest version of the gloves uses contacts placed on the finger tips and palms to tell when the fingers are bent. The other common glove type uses so-called "bend sensors" to detect the position of the fingers. These devices are typically linear devices, giving values across ranges, e.g. from 0 to 127. A bend sensor type of glove can be seen in Figure 6.5(b).

6.1.3 Software

This subsection gives an overview of the basic support that has come to be expected of a VR software system. It is important to note that this is not a survey of VR software systems and will deal with only the basics. A survey of VR systems can be found in [BJ98]. While the survey is somewhat older, it remains the most current survey and remains mostly accurate. Details to systems of higher-level support for interaction and dynamics can be found in the next section, Section 6.2. In this section the basics of VR software systems will be given on hand the VR Juggler framework [BJH+01]. VR Juggler is used as an example both because the authors' familiarity with the system and because it is used as the primary system for the implementation in Chapter 7.

The most basic goal of a VR system is to provide an abstraction from the hardware and low-level system details. This includes all the different hardware solutions we saw in the previous section. The low-level system abstraction includes handling of the setting up windows, viewports, and most importantly the projections, which are necessary for VR to work properly and are complicated to create correctly. The VR system handles the difficulties of dealing with system command for windows and viewport definitions, with differing levels of grace. In the case of VR Juggler, the maintainer of the VR hardware defines the configuration files for the VR systems and the end user/developer must only specify the correct configuration files as run-time argument to their application. This abstraction allows programmers to worry solely about their environment. Possibly more important is that the system removes the need for programming and understanding concepts necessary for the head tracked, position sensitive display of imagery.

The hardware abstraction from inputs to the system varies widely from system to system. A basic abstraction from tracking hardware and software can be expected. Certain characteristics can be expected from the abstraction layer over the hardware and the software drivers. Hardware components typically consist of two possibilities, sampled continuous devices and discrete components. The head tracking component, which is vital to VR systems for projection, is the classical sampled continuous input. The classic discrete input (and ubiquitous) is the button found on almost any input device.

VR Systems are highly oriented to the visual output system. As such, VR systems are based on the underlying graphics generation cycle. The typical cycle of a VR system is outlined in Figure 6.6. As can be seen, the flow of the VR system is tied to the graphics *frame*; the frame being one complete cycle. For effective VR, frame rates of greater than 15 fps are desired [SC03]. This brings complications to the inputs

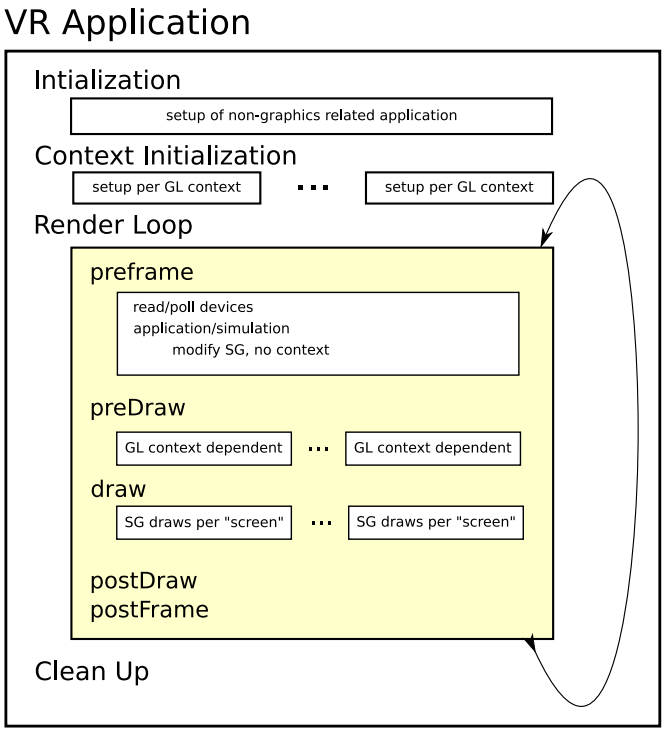


Figure 6.6: The typical flow of a VR system. The naming used here is fairly common, but not all stages are always user accessible.

mentioned previously. Typically inputs are polled at a particular point in the cycle. In most systems, this is directly before the user can perform their application calculations, the preFrame in VR Juggler's terminology. With this method, one ends up polling both the sampled continuous inputs and the discrete inputs. This method inherently introduces latencies into the system.

Another software consideration is that of networked situations. This is required for both clustered computing and also distributed VEs, where participants in a VE are located in more than one physical location and meet in the same virtual location. In both cases each computer has to have identical graphical information displayed, likely from a different position and perspective. For instance in clustered solutions for IPTs, each of the displays has to have a unique view transformation (off-axis projection) that is calculated from the user's position. Naturally, all of these must be calculated with the same position for that frame.

The more difficult networking portions deal with simulation in distributed environments. Anything that is dynamic has to run with the same dynamics on every distributed environment. The most common solution is to have a master computer that controls the simulation. This computer calculates the new environment every frame and then the new information is distributed to the other computers. The AVANGO VR

system is designed specifically to handle this automatically [Tra99]. Some VR systems do not provide any networking support at all. Other systems, like VR Juggler, provide specialized mechanisms for clustered displays, but require extensive programming on the developer's part.

For the connection of the various input devices that are found connected to other computers, e.g. trackers or device PCs, networking protocols are required. A pseudo-standard has emerged within VR, the Virtual Reality Peripheral Network (VRPN) [THS+01]. It transfers input data from various devices to the VR system, with an abstraction level built in. Most VR systems provide interfaces to VRPN as well as most commercial input devices.

6.2 Support in VR Systems

In this section a review of the support for DIVEs delivered by VR systems is provided. The review will be held to a high level, providing only the basic concepts of those systems as they relate to this work. The existing works can be grouped by the general methodology that they use to achieve this. The most prevalent approach is the use or integration of a dataflow system. Standard constraint based methods or simulation systems have also been integrated to include support for DIVEs. Actor and Agent based approaches have also been developed. Finally, a few works have focused more on high-level systems, either by providing a building block type architecture or for the control of complex environments using specialized state machines.

6.2.1 Dataflow Systems

A popular approach to providing advanced support in the VR community is the integration of a data flow system. In most cases such systems are seen as an orthogonal graph to the graphics system, though they commonly intersect. That is the nodes of the data flow graph are often also nodes of the Scene Graph. Data flow systems are an approach well known out of other communities, particularly the control systems and embedded computing area. The approach focuses on describing and implementing the functionality in terms of how data moves through the system.

Data-flow systems typically are composed of two elements: node and connections. There are different possible interpretations of these components, but in the VR community it is almost always assumed that connections are simply transports of values from one node to another. The nodes are the functional components, those that change values. They are also generally considered as black boxes. The functionality of the VE is achieved by specifying the flow of data through this layer. The nodes that are placed in the flow process the data, such that the desired effect is achieved.

There are two basic ways of describing the systems, putting the emphasis on different components. In many ways, the choice taken effects the usage of the system and the design approach taken by developers. The most common mentality in the VR

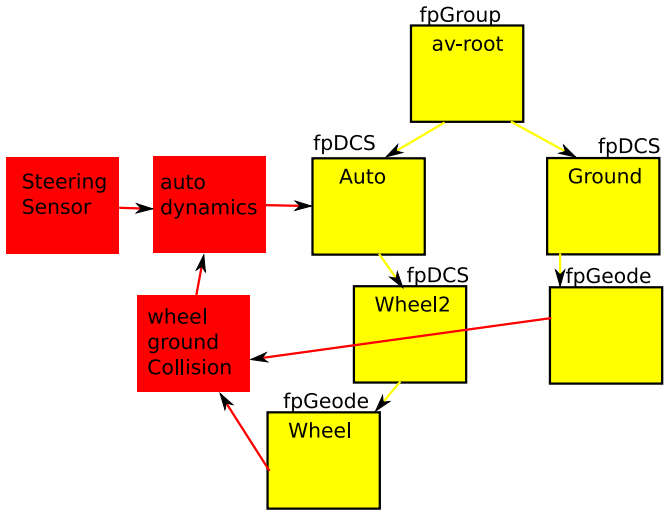


Figure 6.7: An example of a data flow system that is integrated with the scene graph is shown. The example illustrates how an auto simulation would be integrated in the AVANGO VR system.

community is that the connectivity (flow) of the data-flow system is what describes the system. The other description paradigm puts an emphasis on the node components; in these cases they are typically referred to as "filters." This approach is more tends to be more closely related to the control systems area.

Figure 6.7 shows an example data flow graph that is intermingled with the scene graph. Because there are a number of systems that are of interest in our context, the survey of those systems is presented in parts below. The first system is an example of the control systems inspired works, where the focus lies on filtering values. The second part introduces the VEDA system, a pure data-flow system but with unique characteristics. The final part introduces the typical VR approach. This approach has been taken by numerous systems and will be more thoroughly explained.

OpenTracker Input Filters

The OpenTracker system by Reitmayr and Schmalstieg is a data flow system that places the emphasis on the nodes of the system [RS01]. OpenTracker is, however, not truly designed for DIVE creation. Instead it was designed to ease the development and maintenance of hardware setups in a flexible manner, particularly in multi-modal settings. It can be considered an interaction framework, but it is really designed to handle the creation of higher level input data from raw sources. Its functionalities are heavily drawn from the control systems area as it strives to perform the same functionalities. For instance a basic filter in OpenTracker might be a band-pass filter or a Kalmann filter. Another emphasis of their system was on the merging of data from different sources (a.k.a. data fusion), particularly multi-modal sources.

Virtual Environment Dialog Architecture

Steed's dissertation work was based on a data flow system for the creation of his Virtual Environment Dialog Architecture (VEDA) [Ste96]. The underlying system is follows the data flow method as described previously. The focus in building systems is primarily on the connections, but not to the exclusion of discussing the filter creation. However, what makes his system interesting is that it focus is on two different aspects. Firstly, the focus of the system is primarily on interaction methods, but at a much higher level than OpenTracker. Secondly, VEDA is designed to be an in-world tool for designing the environment.

The basics of VEDA's data flow system are four components: devices, interfaces, tools, and the environment. The devices are sensors to the outside world and deliver input from the user. The interfaces are filters that change and process the data generated by the devices. The tools take the output of the filters and apply them to the environment, i.e. the Virtual Environment. Steed presents various filters, including gesture recognition filters and animation filters. An interesting set of filters introduce constraints. These are the type of constraints discussed below in Section 6.2.2. An example is constraining selected objects to the position of an input device. This constraint is a simple way to implement classical direct manipulation.

Perhaps the most interesting aspect of VEDA is that the dataflow structure can be created on-line. A visual programming approach allows the user to connect the components at run-time. The filters/components have to exist beforehand but can be added to the network and connected at run-time. This creates interesting possibilities both for rapid prototyping and also for ease of programming for non-experts.

Inventor Family of Systems

The data-flow approach has been a popular way to extend the basic functionality of the VR system to higher levels. Most of those systems, however, share a single point of inspiration, a system called Inventor. Although each of these systems is unique, they for the most part follow the lead of SGI's Inventor [WG93] system (it was later renamed OpenInventor). This class of systems includes VRML [BPP95], VRML2.0 [ANM97], X3D [BD07], AVANGO [Tra01, Tra03, Tra99] (previously known as avacado), and Lightning [BLRS98]. As this paradigm is so wide spread and one of the implementations, AVANGO, will be used in this work, the methodology followed by this class of systems is explained in a bit more depth. The paradigm will be introduced largely based on Inventor system. The book *The Inventor Mentor* [WG93] provides insight into the design, regardless of implementation used.

Inventor's interaction and behavior support is provided by a number of different components. The basic components provided by Inventor for developing interactions and behaviors in the environment are: Events, Sensors, Engines, Draggers, and Manipulators. Events, Sensors, Draggers, and Manipulators combine to provide a mechanism for interacting with the environment. Engines implement (time dependent) behavior

support. The most critical part of Inventor's design is the Fields and Field Connections. These are the mechanisms that create the data flow nature of these systems.

In Inventor, objects are equipped with *fields* instead of regular member data. The reason for using fields instead of regular member data is three fold. The first is that it provides consistent set and get functionalities. The second is that it provides Inventor with a mechanism for detecting changes in the object state. This makes all objects change aware; using this mechanism they can react to changes as they happen, and only when they happen. The third is that Inventor uses the fields to create smart connections between themselves and the value of fields in other nodes. This is done via *field connections*. Field connections work together with the fields to propagate changes to the values, such that both values are kept the same.

The fields, field connections, and the nodes that contain the fields together form the data flow graphs of the systems. In these systems, the nodes all are created such that their data is stored in fields. Internally, the nodes are also aware of changes to the fields and can update their internal state. This internal state change may involve a simple setting of a parameter, extensive calculations, updating of another field's value, or a combination of these. In this way every node is potentially a part of the data flow graph. In these systems, there are special nodes that are also part of a scene graph system. By creating field connection - usually one way - between the nodes the values are kept in sync forcing data to flow through the system. The nodes previously listed are special purpose nodes of interest in our context and are explained in the coming paragraphs.

Draggers and *Manipulators* are nodes in the scene graph designed to respond to events. They take events such as mouse clicks and movement and create the ability to select and move objects. Draggers react to user events by transforming nodes (via their Fields) in the SG based on user input. Manipulators are specialized nodes that contain built-in draggers. The manipulators are special in that they insert extra objects in the scene based on the input of the user, in order to carry out manipulation. For example, a BoxManipulator adds widgets that are then "dragged" by the user.

Sensors are one place where the systems vary. In Inventor sensors are responsible for creating events based on changes to the environment. Sensors are either time based or watch for changes to the VE. In most of the rest of the systems, sensors are the part of the system that *senses* the external world. Instead of creating special events that are then handled by draggers, they are simply special nodes. They execute before other nodes and only contain output fields. In this way they integrate directly into the data flow system and are only special in the timing of their execution.

For programming behavior into the environment, Inventor provides *Engines*. These objects have at least one input field and at least one output field. Any time an input field value is changed the object updates in output field. The most common input field for engines is a "time" field (see the discussion of time below). In essence, engines are simply a filter node that is explicitly not part of the SG. Engines provided by Inventor include: arithmetic engines (such as interpolators, transforms, and compositions),

animation engines, and event triggered engines. These engines can be used to create various functionalities, such as making a node follow another or animating geometry.

The other major change is that most of the systems have included a scripting language into the system. In VRML2.0 this was in the form of scripting nodes. These were essentially callbacks into the scripting language to allow computation here. Many of the other systems include this functionality, but additionally add a layer of control from that language. For instance, the AVANGO VR system use scheme as a scripting language [STF00]. Callback nodes exist for it, but the main functionality is to build the VE functionality through scripting. Nodes are created and added to the SG as well as creation of the entire data-flow system via field connection creation. Beyond the ability to script the environment and its behavior in an "easier" language, these systems use run-time interpreted languages.

Although there are various support mechanisms described above, the actual implementation of time dependent behaviors is still based on a time signal. As with the Engines described above, a special time sensor is created. The output of the time sensor is either the system clock or the time delta since the last frame. Nodes that create time based behaviors, including interactions, receive only this for assistance in programming the behavior. No further support is given for the time based implementations.

6.2.2 Constraint Based Systems

A few systems have added support for dynamics based on constraint networks [BG95, Del00, ESYAE94, JDM99, TLG99, WGW90]. These systems work by adding constraints to objects, for instance $A - B = 5$. The collection of constraints is then solved every frame (or so often as it changes) to adjust the values to make sure that all constraints hold. Such systems are called constraint solvers. Such solvers also have some rudimentary time based elements built into them, but largely focus on algorithmic relations. For instance, an object should stay 3 meters from another object at all times. However, some of them allow the definition of continuous time functional constraints.

These approaches are quite powerful and expressive. Two of the systems have focused on the inclusion of continuous time functionality, TBAG [ESYAE94] and DLoVe [Del00]. The DLoVe system from Deligiannidis is the most resent of the systems and most advanced. DLoVe allows user input to be a node of the network of constraints. It also has the ability to turn on and off constraints at run-time. However, all constraints must be in the network and only whether they are evaluated or not is changed. This means no emergent (unforeseen) behaviors are possible, such as the addition or deletion of random objects. Three factors detract from the applicability of constraint based systems. Interaction, though included is limited by the method, as all interactions must be foreseen and programmed into the system. The second is that constraint solvers do not scale well (each constraint is another simultaneous equation to solve). This limits the applicability of the system to small environments, particularly when one considers the need to include all possibilities for interaction and behavior. The final point is the usability in terms of programming. Although any behavior can

be defined by the constraints, programming those definitions can be challenging. This is due to the fact that one does not think of most behaviors in terms of constraints.

6.2.3 Simulations

A number of recent works have made efforts to more generically couple simulations with VR. An introduction to the area of simulation can be found in Section 3.4. Many simulations have been performed in VR, but typically systems have either been special purpose (e.g. flight simulators) or were limited to a playback system that viewed the results of the simulation. A few recent efforts have focused on coupling existing commercial simulation systems with VR systems in generic manners. A good overview and analysis of those efforts can be found in Strassburger et al. [SSLR05]. These focus on incorporating discrete simulation or discrete event simulation in "temporally parallel" manners, such that the visualization time is synchronized with the simulation time. The difficulties and ways of integrating these systems are explored there.

In that work and also in Rehn et al. [RLVD04], they discuss the creation of simulations that allow interaction by the user at run time. This places further constraints on achieving a temporally parallel system, as the simulation has to be fed the input from the system, such that it can reacts without introducing lag. Rehn et al. present two interactions typical of many simulation systems in VR. The first is the ability to change the flow of simulation time. The second is being able to get current data from the simulation. Even with such coupling, interaction seems to be severely limited. In the case of Rehn et al. [RLVD04], the interaction described in their example was retrieval of the status of parts on an assembly line, i.e. selecting it and seeing simulated data on that part.

6.2.4 Entity, Actor, and Agent Based Systems

Another direction adapted by a few early systems is that of entity based systems [CH93, Has96, KT99, TC95]. These are directly related to the actor systems of the virtual humans (sometimes called avatars) researchers. Although there are variations involved in all the systems, the basic concept is to imbed all of the logic of the object into the entity itself. The basic way of doing this is by dividing the functionality that is programmed into two parts. There is the behavior of the object, an internal functioning that is inspired by the logical processing of a human. The second part is a reactive nature. These reactions either cause instantaneous reactions or modify some part of the internal behavior structure.

Typically, the behavior portion of the entity is controlled by a single function that processes every frame. The reactive portion is implemented differently. Typically, a specific function is designed to react to specific events. The function receives the event of interest and is run at this point, (pseudo) asynchronously. Typically, these functionalities are written in a scripting language. The element of time is incorporated by the

authors own inclusion in their work. Modern Virtual Humans may be much more complicated that this, involving multiple layers instead of just reactive and behavior. For instance, emotion and goals often are important to their design. Yet the basic structure is typically similar. Please refer to Magnenat-Thalmann and Thalmann [MTT04] for an in depth look at Virtual Human research.

A slightly different approach has been taken by Geiger [Gei98, GPRR00]. He approaches this from the direction of attempting to prototype VEs quickly. His approach is a combination of agent methods and animations, implemented in actors. Every part of the environment is an actor. Those that change are built up of animations. The animations are of the classical types: keyframing or kinematically specified. The actor concept builds upon these animations in a way that is designed for abstraction and reusability. At the highest level, the actors have the ability to choose between the possible animations and communicate with the outside world. This outside communication is used for communication among different agents, i.e. forming an agent based system. Using this communication, the actors work together to build a scene. For this work he developed the AgentGHC (Guarded Horn Clauses) system to handle the agent implementation. The complete system, as presented in his dissertation, is the Agent Animation Layer (AAL) [Gei98]. The system seems to have been later renamed i4d (interactive 4D) [GPRR00].

6.2.5 High-level "Building Block" Systems

A few systems, most notably Alice and Virtools, have approached the creation of content differently. These have based their approach primarily on providing basic functionalities as building blocks on which more complex functionalities are then built up. Alice is not VR capable, but its approach is interesting [PBC+95]. Alice is extremely simplistic in what it provides, but its strength is that it can be learned very easily. It is designed, at least in the second and third versions, for learning to programming and has been used to teach high-schoolers and even grade-schoolers. The user builds behaviors up from the most basic functionalities, like rotate for X steps.

Virtools is a commercial VR system that follows a similar approach, but is more powerful and designed for VR usage [Vir]. A large library of functionalities is provided for building up complex worlds and complex functionalities. When the functionality required is not present, the user can program new functionalities. The programming is done in the C++ language. To assist in the creation of time dependent behaviors, the user is provided the time delta between frames.

6.2.6 State Machines

The approach of using state machines and their ilk has been applied in VR mostly at the highest level of the environment control. Few systems for support have been published using this, though this approach has been used often, for instance in the area of Interactive Storytelling [BLM+04]. One system of note builds a system of

support in VR using an approach out of this category, again for high-level environment flow control. This system is composed of "Hierarchical Concurrent State Machines" (HCSMs) [CKP95]. The HCSM system was designed in the research context is of vehicle simulation. Specifically, the system was designed to control an environment that the user (of a motion platform driving simulator) was navigating. The system had to control all the stop lights, other cars, bystanders, etc.

HCSM extends standard state machine systems by moving to a "communicating, hierarchical" arrangement. Instead of nodes being the basic component of their system, state machines are the basic component of an HCSM. Since there are multiple state machines in the system, many states can concurrently be active. Each of the running states provides information for some part of the simulation and the same component in the VE may have multiple running states. Extra functions outside of the state machine determine which of the outputs are sent on and which are merged with others.

As with all such systems, the control of the actual dynamics is controlled within a single state. In HCSM a single "activity" defines what happens. The changing of states provides a method of changing which behavior is active. In HCSM and in the general state machine case, the support for the actual implementation of time based semantics is limited to the support provided by underlying programming language.

An additional point of interest is that the HCSM system extends the basic state transition. This is done in two ways, recognizing that the instantaneous transition between states does not always create acceptable simulation. The first is that each state can have *entry* and *exit* functions. These functions are called during the transition in or out. The second is that special *transition* functions can be placed into the space of movement from one state to another.

6.3 Efforts to Model DIVEs

Alternative to the creation of support systems, a few efforts have sought to model or specify the content of Virtual Environments. These works are largely VR system independent, though not always completely. They investigate how to define and specify the environment that is displayed in the VR system. Adding behavior and interaction to objects are recurring themes in these works. However, in general, they present little in terms of concrete ideas on how to achieve these goals. This section presents those works at a high level, focusing on the concepts present rather than on details of the specifications and models.

6.3.1 Specification and Modelling Languages

Several works have investigated methods for specifying or modelling VEs. In particular, these works have focused on VEs that have been here referred to as DIVEs. In this section, the most relevant of those works are highlighted. The presentation of those

works focuses on the components that are interesting in the context of this dissertation, rather than providing a complete summary.

The VRID model [TJ01] for Virtual Reality is interesting because of several of its ideas. This work develops a process for defining the VEs, based on a multi-level approach. In the high level design phase, the data elements, objects, and high level specification of components are defined. In the low level specification the actual details of the components are specified. VRID's modelling of the objects was defined five components: graphics, behavior, interaction, mediator, and communication. The graphics component specifies all relevant graphics details. Behaviors define what objects do. To simplify the modelling, the further specify that behaviors may be physical, magical, or composite (combinations of physical and magical). Mediator components are responsible for merging conflicting behaviors. The communication component is responsible for all external information exchange.

In moving from the high-level description to the concrete behaviors, VRID did a few things of interest. One is to explicitly suggest that the behavior is a mixture of continuous and discrete time. Continuous behaviors are defined in a data-flow style system (actually a precursor to the DLoVe constraint based system presented in Section 6.2.2 called PMIW). The discrete nature of behavior is captured by state machines transitions. VRID is unique in that it suggests how to proceed through the process of designing a world. This is not just the high-level/low-level split, but they suggest a process to use in both levels.

An alternative method for modelling VEs was presented by Zachmann [Zac96]. Zachmann proposed the *AEO triad* to define the world; that is, the world is defined by: Actions, Events, and geometric Objects. Actions represent anything that can "happen" in the VE. This includes interactions, behaviors, and changing properties. Events are triggers to cause actions to occur. A grammar based on this is built to describe the properties of the environment. An example of a statement in that grammar is, "animation switch on whenever button is pressed down" [Zac96]. Zachmann's work is theoretical and does not suggest how to actually implement such a system.

The Web3D community has taken up the topic of specifying more complex and compelling VEs. Various papers over the topic have been presented [DR03, MH06, Sto06]. The overarching goal of the presented methods is to simplify the creation of interactive dynamics, particularly for non-programming users. In general, the approaches pursued follow Zachmann's approach of creating a high-level abstraction. Both [DR03] and [MH06] propose the extensions to the X3D specification to include schemas that describe the high-level dynamics and interactivity. The implementation of this is highly dependent on the Inventor style of data-flow system presented in Section 6.1.1.

Pellens et al. in [PTBK05] approach the definition of VEs from a different direction. They propose using ontologies to model environments. The methods they introduce encapsulate the object and behaviors in powerful semantic ways. However, the "behaviors" they define are limited to three types, move, roll, and turn. Such functionality is similar to the first version of Alice [CAB+00] (also described in Section 6.2.5).

To combine them, they introduce operators that can implement Allen's temporal operators (described in Section 3.1.3). Although for the purposes of this dissertation the approach is ill-suited due to its lack of actual implementation assistance, it could potentially be a complementary approach to the one developed here.

6.3.2 Temporal Modelling

Campos et al. proposed a temporal model of VR objects in [CHE02]. This model represents the semantics of the changes to objects. By representing the behavior of objects at a high level, they believe that users can perform spatio-temporal reasoning about them (see Section 3.8 for information about temporal reasoning). The model they proposed is based on an interval based view of time. Core to their proposal is keyframe animations. Four components build their approach: normalized acts, timed acts, courses of action, and animation space.

The basic definition is a normalized act. This is a normalized (time 0-1) keyframe animation. This normalized act is part of a motion retargeting scheme (see Section 3.3 for references), where the timed act is the normalized act stretched to the actual time required for the animation. The interval of the timed act must be determined before starting it. They propose different methods of retargeting the keyframed values in [CEH03]. To define the behavior of an object, they define courses of action. These define the temporal relation of timed acts. Here, they include concepts from Allen, such as join and equal (see Section 3.1.3). At this stage temporal reasoning can be employed on the system. Animation space is the temporal ordering of courses of actions. Finally, they combine "VR Objects" with courses of actions to creation what they refer to as an "Animation."

In describing the semantics of VR Objects (though at this point the description seems to include the animations) they introduce an idea of history. The history of the object defines what has occurred to the object and also what will occur. Here, they note the difficulty of introducing interaction into the semantics they have developed. Although their treatment of the issues is not complete, they provide examples of where some of the issues lie.

6.3.3 VR as a Hybrid System

Smith and Duke take a approach to defining VEs that is different from others, at least in its explicitness. They propose to handle VEs as Hybrid Systems (see Section 3.7 for more on hybrid systems). In [SDM01] they focus on interactions in VEs, noting that interactions are both continuous and discrete based. They note that the user's view of the interactions is continuous even if the actual implementation is otherwise. In [SD99], they introduce a modelling method based on Petri Nets. They use a high-level Petri Nets system called HyNet, which is an extended Petri Net to better model hybrid systems (similar Petri Nets are described in Section 3.6.1). This system allows them to model portions of the system as continuous in nature and still include discrete

transitions. The details of how this system works can be found in [SD99]. For the actual implementation of time based components, the proposed system provides no extra support.

Chapter 7

Functional Reactive Virtual Reality

A new system to support the design and implementation of Dynamic, Interactive Virtual Environments is developed in this chapter. The design of a system supporting DIVEs is first explored. This is approached by setting use cases and requirements. A discussion of the applicability of different approaches to meet the requirements is then given. Based on the design discussion, a system is proposed based on the Functional Reactive Programming paradigm.

The developed system couples the Yampa Functional Reactive Programming system with existing VR frameworks to make a complete base system. The resulting system has taken its name from this combined nature, Functional Reactive Virtual Reality or FRVR (pronounced as the word fervor). This chapter introduces the basic solution for achieving a system of support based on the existing technologies, Yampa and VR systems. Only those developments required to achieve this coupling are presented here, including a number of developments necessary for a minimally working system. The next chapter will introduce extensions to the FRVR system that are necessary to fulfill many of the requirements for the creation of DIVEs identified in Part I.

The first section presents the design of the system. In section 7.3, the implementation of a system based on that design is presented. The work flow that the developer has using the system is explained in Section 7.5. In Section 7.6 the resultant system is evaluated with respect the requirements developed in the design section.

7.1 System Specification

The design of a system to support the creation of DIVEs is developed in this section. In the first portion, use cases, of the system are collected to inform the development. Based on the use cases and the developments of the prior work on the nature of Dynamic, Interactive Virtual Environments, a set of requirements for the proposed system are

presented. A discussion of the possible approaches that could be taken is given in the final portion of this section.

7.1.1 Use Cases

To inform the design, a number of use cases are identified. These use cases come from a number of different sources. The first use case, though with a broad interpretation of the term, is the initial impulse for the dissertation's topic, enabling a wider community to be able to develop DIVEs. Section 7.1.1 collects the use cases developed during the investigation of the Dynamics and Interaction spaces of DIVEs.

Enabling a Broader VR Community

The direction of this dissertation is driven by a number of issues the author witnessed in working with people desiring to develop DIVEs. Those observations spurred the decision to work on the issue of supporting DIVEs. The overarching impulse was to develop a system of support that better matches the human understanding of the dynamics of an environment. The frustration of various programming savvy engineers attempting to creation dynamics in their VEs started this thought. Due to the way computers work, classical computing is described as series of discrete steps. In the context of VR systems that only produce 30 frames per second, this means the discrete steps are large. However, they are also inconsistent in length, some close together some far apart. Yet, the human does not interpret the world around them as being a series of discrete occurrences. We propose that this discrepancy is part of what makes programming DIVEs with current systems difficult.

Exactly how humans perceive the world around them is a point of contention. The time aspect of this has been introduced in Section 3.1, where philosophy played a major role. From that discussion, we can assume that people perceive the world around as a kind of hybrid system. There are continuous actions that take place over time and a few discrete events (most of them in relation to technology). Often, those events are what shape the continuous actions that are occurring, at least in human perception. Even as babies, continuous phenomema are understood before events [vP05]. In the majority of systems, people who have an understanding of the world around them that involves a mixture of continuous activities and events are asked to program with a system that is defined by time stamps on execution (which occurs approximately every 30th of a second). Closing this gulf of understanding was the initial impulse for the system to come out of those experiences.

Those experiences exposed another factor to the difficulties experienced by the potential developers. Many of those developers were not VR experts, but instead from one of the various fields identified earlier that want to use VR as a tool for other research purposes. Many of these potential developers were familiar with programming and often even having specially taken courses in programming in the language used (C++). Nevertheless, even the engineers who programmed often struggled with the creation of

their conceptualized dynamics and interactive dynamics. Often they managed only a minimum, essentially giving up on the task.

Ideally, a system that is approachable by non-programmers should be developed; however, this goal has been set in many of the systems developed, for instance Alice (see Section 6.2). This is hard to achieve in a VR setting and with the flexibility and power that allows users to create any world. For instance, Alice is highly restricted in dynamics (simple forward dynamics programming like "turn for 2") and is not VR capable. Pursuing such a system ends up restraining the system too much or narrowing the focus to a very specific area of application. Instead, a system that better supports an expanded scientific community is chosen to be an acceptable compromise. At minimum, those with some programming experience and with strong mathematical backgrounds should be able to use the system. For instance, Mechanical Engineers that are used to simulation software would be a target group for the software. They are intimately familiar with simulation, but the constraints are very different on such systems and the style of VR programming is difficult for many of them to adjust to.

DIVE Use Cases

The following use cases from the area of DIVEs were been developed in Sections 4.3, and 5.1.5:

Path Recording/Playback One often desired functionality is the recording and playback of paths taken through the environment. For instance, it is desirable for demonstration purposes as well as review of experiences in training applications and in therapy sessions.

Creation/Deletion of Objects in a Scene The creation and deletion of objects in the scene is a classical dynamic of VEs. Adding Support for it is more challenging than other simple state changes in the Scene Attribute category, since it is its purest form it is a dynamic that requires run-time allocation/deletion of objects.

Spatial Behaviors The most basic dynamic of the visual environment is something moving spatial through the environment.

Inter-object Changes Showing that subparts of an object can be independently controlled from the object as a whole demonstrates the level of control of the system.

Entities The ability to create entities demonstrates a higher level of functionality and logic.

Story A abstract level of dynamic should be shown through a demonstration of storytelling.

Time It should be shown that time itself can be manipulated in the VE.

The area of Dynamic Interaction contributes two additional use cases:

Direct Manipulation Continuous direct manipulation of objects should be demonstrated.

Higher-order Direct Manipulation The continuous direct manipulation of an object, such that the system derives the velocity of the user input as part of the interactive nature of the system should be demonstrated.

Finally, the area of Interactive Dynamics did not develop any specific uses cases, as the requirements listed define well the different usages in themselves. As such, the development of variations of the interaction with dynamics should be explored and shown.

7.1.2 System Requirements

An analysis of the system requirements is a necessary endeavor for the development of a valid and usable system. In this section, requirements will be collect and presented. The presentation of the requirements begins with those requirements that come directly out of the problem statement and motivation of this dissertation. Next, a set of requirements are placed on the system, based on achieving a good software structure. Finally, the requirements developed as part of the investigation of the Dynamic, Interactive Virtual Environments design space are collected together.

Requirements Derived from the Dissertation Intent

A number of requirements on the system are products of the impulse that lead to the definition of the dissertations theme. Those motivations are included in the introduction, but also to large part in Section 7.1.1 above. Rather than repeating that information, the requirements that result from those areas will be presented here. In addition to these factors, the method used in system development was restricted already in the definition phase, based on observations of the development of their systems. Before collecting the requirements, that observation and its impact will be discussed.

Many systems for support of DIVEs or similar goals are tied to a specific VR system. By tying the development to a single system, the community that benefits from the system is strongly reduced. Often, systems are built into proprietary, or at least non-open, systems. In some cases complete new systems have been built to create DIVE support. In addition to restricting the community by this coupling, a system that is instead designed to work across multiple VR systems has a number of benefits. The monolithic structure that limits the lifetime of a system in the changing world of graphics is to be avoided. If any one of the systems becomes obsolete or no longer available, porting to a new system should be easily performed if the system is properly designed. The potential community for the software increases when more VR systems are supported. Finally, avoiding the need for programming a VR system

alleviates extensive work. A number of systems that have been proposed have become so entangled in the implementation of the basis system that the support of the actual goal is minimal.

The requirements that come out of the motivation for and definition of the dissertation are:

- use a representation of dynamics that better matches our understanding,
- should not make the developer deal directly with δt,
- use a hybrid (continuous and discrete event mixture) time representation,
- be built on existing VR systems, and
- should not be a monolithic design.

Software Engineering Requirements

A number of additional requirements are placed on the proposed solution. These come from the system specification side. These come predominantly from software engineering requirements. However, many are informed by experience with various other systems, in particular VR systems. Requirements from this vantage point are:

- be cross platform,
- be cross VR system,
- be (soft) real-time capable,
- be user extensible,
- be scalable, and
- be releasable as open source.

These requirements are fairly standard for many projects. The cross platform, cross VR system, and open source requirements are general requirements, but specifically important to allow a wider community to be able to use the software. The open source release of the code also assures that the development can further grow and be adapted in the future.

Three further requirements are to be considered, but are of a lower priority. They are meant to be incorporated if possible, but not to unnecessarily limit the design. These requirements are:

- provable code,
- distributable, and
- concurrent multi-core/multi-processor capable.

The ability to prove that the code does what it is supposed to is of importance in some areas. Traditionally, military agencies want to be able to assure that programs do what is expected. Similar cases exist in other applications areas, where the simulation

should prove some design effective. Until recently this has highly influenced the choice of languages used and narrowed the scope of infiltration of VR in the field. A system that supports this at the application level would be advantageous in verifying the validity of the implementation.

Distribution is a classic part of VR systems. In light of applications like MMOGs, having a design that does not hinder distribution is beneficial. Any design that supports distribution is good for future expansion of the system. Related to this is the ability for the code to run on multi-processor and multi-core systems. For expansive and complicated environments and simulations this can be the difference between success and failure of the system.

DIVE Requirements

The requirements on the system necessary for supporting the creation of Dynamic, Interactive Virtual Environments were developed in the analysis of Part I. These were divided into four sub-areas: Dynamics, Interactive Dynamics, Dynamic Interactions, and Dynamic Interactions with Dynamics. Since the requirements were each developed in the first part of this dissertation and complete discussion of the points was provided in those places, the requirements will simply be collected again here. References to the places where they were developed are provided.

The requirements for generation of Dynamics as developed in Chapter 4:

- support for time representations and implementation of
  - continuous
  - continuous over interval
  - discrete time
  - ordered set of intervals
- support for sets that are order determined at run time
- support for run-time changes to the VE
- ability to manipulate time, including the ability to
  - freeze time
  - change itself, i.e. time's "speed"
  - define multiple time "speeds," potentially per dynamic
  - undo, including over all time representations (where possible)

Dynamic Interaction and the developed requirements were presented in Section 5.1. The resulting requirements were:

- support for time representations and implementation of
  - continuous
  - continuous over interval
  - ordered set of (continuous) intervals
  - ordered set of discrete events

- enable handling of input
  - including all values, not only the most recent
  - including support for writing filters, in particular time based ones
  - supporting direct manipulation interactions
- support for inferring higher-order functions of time on continuous inputs, e.g. velocity and acceleration
- support for multiple output modalities, with different update frequencies
- support for hierarchical levels of Dynamics Interactions, for instance for conversation, which is respectively composed of speech interactions.
- provide an ability to record and play back Dynamic Interactions
- provide an ability to undo Dynamic Interactions

The required support for Interactive Dynamics as developed in Section 5.2 includes:

- existence of events
  - generated by external inputs (e.g. button presses)
  - generated internally (e.g. collision of hand and bee hive)
- reaction to events by
  - stopping dynamics
  - starting dynamics
  - freezing dynamics
  - continuing frozen dynamics - no time passed
  - continuing frozen dynamics - as if they had run

Finally, the requirements to enable experimentation with Dynamic Interactions with Dynamics developed in Section 5.3 are:

- in/out functions
- blending
- user defined transition functions
- exchangeable (interpolation) kernels
- basic sets of interpolation kernels (linear, sinus, etc)
- support for different timings of the functions

For a more in depth look into the individual requirements and how they were developed please consult the individual chapters covering the topics in Part I.

7.2 Design

This section develops the design of a system to meet the goals and requirements set forth. A basic approach to be used for the system is selected in the first portion. In the second portion of the section presents the possible system architectures that could be used based on that basic approach. In the final portion of this section the completed design of the developed framework, Functional Reactive Virtual Reality is developed. The details of the developed framework are presented in the following sections.

7.2.1 Selection of a Basic Approach for the System

The basic approach to be taken in the design of a system is discussed here. Informed by the use cases and requirements collected in the previous sections, the potential design approaches are revisited here in regards to their applicability to the highlighted criteria. Approaches considered are those presented in Chapter 3, which focused on general time dependent approaches common to Computer Science, and in Section 6.2, which covered the existing approaches taken in the VR community.

The dissertation's definition itself strongly limits the number of potentially applicable systems. The real-time requirement for the VR context is critical; although, a soft real-time strategy is acceptable. Meeting this requirement significantly reduces the number of applicable systems. Many of the approaches out of other CS fields fail to meet this criterion. The various time representations required for DIVE creation further restrict the selection. The combination of the continuous and discrete representations essentially restricts the approach to those for hybrid systems. Considering just these two requirements, four potential approaches can be identified: a hybrid simulation system, a constraint based system, a physics simulation, and a custom built solution.

A complete physics simulation traditionally would be thought impossible for such a task, due meeting the real-time constraint. However, the introduction of various hardware acceleration cards [Gee06] indicates this solution has promise, as they are designed primarily to provide physics simulation for computer games. However, simulation based on physics is difficult to adapt to many of the needs identified for DIVEs. Particularly in the areas of Dynamics and Interactive Dynamics, a large portion of the design space would be difficult, if not impossible, to program in this way. For instance, the implementation of singular entities in a physically based way is only feasible for their movement, and, even then, it would be taxing. Using classical physics to describe the behavioral level of an entity would be daunting.

Constraint based methods have been developed for VR previously [BG95, Del00, TLG99] and are discussed in Section 6.2. Constraint based methods match many of the requirements laid forth. The systems previously developed demonstrated their capabilities. Typically these systems lack run-time flexibility, i.e. all behaviors must be preprogrammed and also evaluated. However, Deligiannidis' DLoVe system demonstrated it is possible to have a run-time modifiable network. This was done through a special mechanism that allowed branches to be switched on/off, allowing a dynamic sub-set of the constraint graph to be selected for execution. Two further factors influence the feasibility of using the constraint networks approach. Firstly, constraint based systems have problems meeting real-time constraints when scaled up to larger systems. For small systems they work, but they are computationally intensive, even with modern solvers. Secondly, the approach suffers in regards to the usability and to matching the developers understanding of the problem. Constraint methods require the user to think "backwards." Essentially, the developer specifies not what is happening, but builds a set of reverse equations that allow the solver to enforce the proper dynamic/interaction.

The remaining potential approaches are hybrid systems and custom design. Usage of existing structures is much preferred, as examples of custom solutions show that they tend to become monolithic structures and narrowly focused in application scope. Hybrid Systems are generally custom solutions in themselves, at least as far as that research community supports them. Instead, the community focuses on the modelling and verification of Hybrid Systems, as explained in Section 3.7. Application areas that fall into this area, such as embedded systems and robotics, tend to prefer hand coded systems rather than general frameworks. This is primarily in order to meet the hard real-time requirements of the fields. However, one system out of this area exists that can fill the basic requirements and has development potential for the remaining requirements, Functional Reactive Programming.

Functional Reactive Programming (FRP) is a language for programming hybrid systems; although, it is not generally thought of as such within the developing community. Pembeci et al. address this facet of FRP in [PH03]. Appendix A introduces the FRP concept and its implementations in depth. FRP is based on two components, continuous behaviors (functional) and discrete events (reactive).

The continuous behaviors are based on integral mathematics, making it a good match to the requirements outlined in regards to time representation and also modelling behaviors as they are understood. The event system of FRP is explicitly developed for the creation of its reactive nature. The system reacts by using a variety of methods that switch the active function to a different function. However, events are simple data that either exist or not and can have values associated with them. This means they can be used for other purposes as necessary.

The impulse for the creation of the FRP paradigm, was to create a more natural way to define computer animation [EH97, Ell98]. As with our conjecture, Elliot felt that animation methods, mostly keyframing at that time, were suboptimal, as they did not match the users understanding of the movement being created. Elliot felt that the user perceived the movements as continuous actions and, therefore, developed a system based on that. The basic of the system being integrals with respect to time, i.e. continuous functions over time. This enabled simple implementation of animations using forward dynamics.

The real-time ability of FRP and even its underlying Haskell implementation language have been demonstrated through a number of applications built with it. Section A.5 discusses some of those applications. Particularly the robotics and game applications show that FRP can be used for (soft) real-time applications.

Perhaps one of the biggest benefits of FRP is the reactive nature built into it. This reactive nature introduces a very viable possibility for interactive dynamics. Investigating those possible interactive dynamics presented in Chapter 5, many of them fit well into a “reactive” framework, that the interaction triggers something new to happen. Although rarely discussed in the FRP literature, dynamic interactions are also easily implemented. Since external inputs can be provided to the FRP system, a functional based (here this term refers to the programming paradigm, functional programming,

rather than the behavior meaning of FRP) approach is well suited to their implementations. In essence, the functions need only to shape the values that pass through them. For instance, if the position of an object is being manipulated, the function takes the user input, calculates a new position based on that, and returns the new position of the object.

The current implementation of FRP, Yampa, shows potential for implementation of some advanced features. Yampa is described in depth in Appendix A.4. The "Space Invaders" game implementation demonstrates how the simulation can be a run-time specified, e.g. the number of alien spacecraft that are attacking is variable throughout the game. This is something that other possible systems reviewed have trouble doing. The Yampa implementation introduces the idea of "freezing" behaviors. This seems to have potential for many aspects, such as spatial partitioning schemes.

The largest questionable part of a decision to build on the FRP concept is the usability aspect. The advantage of FRP being built into Haskell, is that it is based on a mathematical syntax. Those familiar with mathematics should find the Haskell code easy to follow. However, the code is quite foreign, both in appearance and in programming methodology, to experienced procedural language programmers (e.g. the VR experts). This is partly overcome by the addition of the Arrows programming paradigm, which allows procedural style programming in Haskell. The original FRP implementations could be dismissed off-hand based on how difficult it was to program even simple examples. However, the Yampa implementation addresses this in at least part. While the code remains different that traditional procedural VR programming, after a period of adjustment the syntax becomes readily understandable.

Given the clearer syntax of the current FRP system, Yampa seems viable for use. It remains mathematically based and builds on the time representations that are required for DIVEs. The alternatives are constraint networks, involving thinking of problems in reverse, and classical VR Object Oriented approaches based on advanced C++ techniques. Compared to the alternatives, it is reasonable to expect that Yampa/Haskell syntax is at least as understandable and would have a shorter learning curve.

7.2.2 System Architecture

The Functional Reactive Programming paradigm was selected as an appropriate approach for the implementation the dynamics and interaction of the DIVEs space. An implementation of the dynamics and interaction components has to be coupled with a method for presenting the world. The requirements specified that the complete system should work in conjunction with existing VR systems to present the results in immersive display systems. In this section, a basic system architecture that couples the FRP system with a VR system in a satisfactory matter is explored. Three potential methods are described, listing their respective advantages and disadvantages. The discussion also introduces a number of implementation details that weigh heavily into the selection of an architecture.

A number of factors play into the selection of a method of combining the systems. On the programming side, the Haskell based FRP system has to be paired with a C/C++ based VR system. The systems cannot be combined directly, as the languages of implementation are not compatible. In the VR community, the typical method of incorporating an external system that controls "objects" of the world is coupling the external system into the SG, which is where the "objects" exist. This method causes a "tight coupling" of the systems. Conversely a loose coupling of the systems could be conceived of. This allows the two systems to be maintained independently. In such a method, they are only connected through an interface that provides for the exchange of values. The independence advantage comes at the cost of introducing to two problems: synchronization of information is difficult and performance degradation.

The loosely coupled architecture can again be achieved in two main ways. The major difference here is on how the synchronization problem is approached. The methods follow either a "lock-step" method or a free running thread approach. The lock-step method synchronizes the two systems such that they run in locked step, i.e. that either the FRP side is running or the VR system. The free running thread approach instead allows both to run independently, i.e. at the same time.

Each of the three architectures identified are explored in the following sub-sections individually. Their benefits and detractors are discussed. Where appropriate, relevant implementation details that impact their feasibility are explored. The tightly coupled method is addressed first. The lock-step loosely coupled architecture is investigated next. Finally, the independently running thread architecture is described.

SG coupled FRP-VR

Incorporating the Yampa FRP system into a VR system by coupling to the objects follows the typical path used in the VR community. Examples of this method can be seen throughout Section 6.2. The motivation to use this approach is founded in the presumption that the scene graph is in control of the objects of the world. Integrating the control of the object's actions into the place where they are defined is a logical approach. When coupling an external system, the most common solution is based on a callback structure. The diagram of Figure 7.1 demonstrates the principle used. This leaves the VR system in control of the object. Control structures for calling the external system have to be present in the scene graph system or have to be integrated in the system in some other manner.

This coupling solution has several advantages and disadvantages. Naturally, the biggest advantage is that there is a clear ownership of the object and, with that, an idea of "where the object lives." This reduces to the fact that all portions that describe the object are co-located, i.e. they are found or directly called from the SG object. For instance, Figure 7.1 illustrates how this would be implemented for a simple pendulum in a scene graph style system. The bottom most scene graph node contains the geometry for the pendulum. The parent node of the geometry is a transformation node (a matrix)

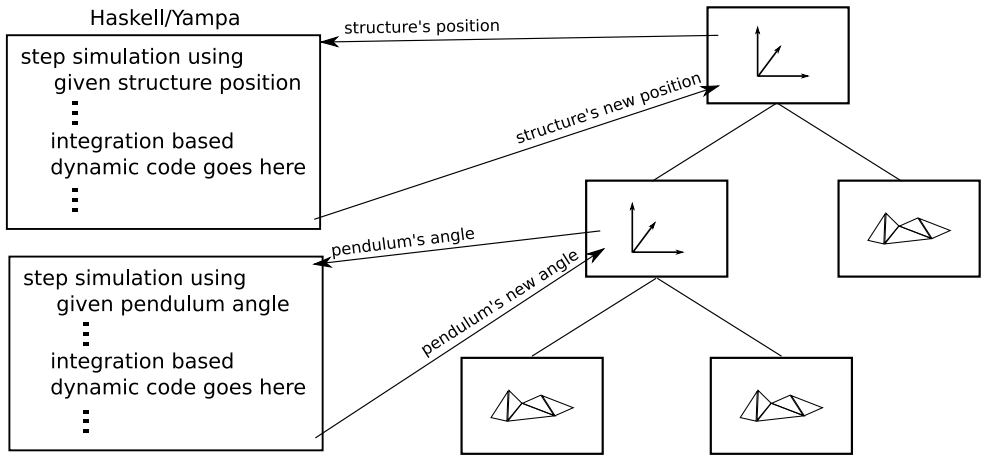


Figure 7.1: The integration of the Yampa FRP system into the VR system by use of a callback structure.

that controls the orientation of the pendulum. This node is an extended node that incorporates the callback into the Yampa simulation.

The advantage of this system is illustrated by the example. The dynamic or even interaction of an object is incorporated into the representation of the object in the scene graph. This merges the semantic meaning of the object and its visual aspect. That the method is the classical approach in the VR community (VR systems such as AVANGO, Lightning, and DIVE do this) makes the method particularly appealing, as it is familiar to VR experts. An additional advantage is established optimization techniques that exploit the scene graph design can be used. Among the most interesting is reducing application processing by only performing dynamics/interaction calculations when the object is spatially local. However, the solution may not be optimal as things just behind the user are typically culled by traditional scene graph methods as well.

Unfortunately, the major benefit of the method, having the semantics integrated with the representation, is not always straight forward. Taking even a simple extension of the pendulum example creates complications; if we consider a Newton's Cradle (see Section 9.2.4 if you are unfamiliar with the Newton's Cradle), some of the issues can be highlighted. In the simple case of a Newton's Cradle, a single ball is set into motion. When it hits the other four balls, the ball on the other side moves into a pendulum motion, until again striking the four balls. Two of the five balls are in motion over time, but not at the same time and, yet, are dependent on each other. The implementation of this is our SG based object setting is more complicated. The Newton's Cradle as a whole needs to be simulated, but two underlying objects have to be moved. Each ball has to have an individual transform so it can be moved, and the Newton's Cradle as a whole would likely have a grouping node over everything. The major question is how the impulse is transferred between the balls on collisions (as well as how the other three balls play into it) is unclear.

This difficulty of actually integrating the semantics of an object into the scene graph representation is not unique to this kind of example. Throughout the discussion of DIVEs in Part I, a great many things that can change over time were identified. Looking particularly at the dynamics design space, it is easy to see that one would like to do more than just control the position and orientation of objects, which is what scene graphs were designed for. Another issue is that the visual representation of parts of the environment is very limiting. The advantage of coupling the semantics of an object with sound/haptic/smell into a visually oriented scene graph is highly questionable. There were also a number of abstract quantities that could be changing over time. Having to provide an object as part of a visually tied system is not optimal.

A further disadvantage to this coupling is that the development is constrained to a specific system. Beyond the fact that we have identified VR system independence as a goal for this system, in a system like VR Juggler there are six different rendering engines available. There are two basic solutions: writing a coupling to every VR/SG system and writing some sort of wrapping layer for abstracting away from the VR/SG specifics. In both cases, new C++ classes have to be written either for each scene graph or for new dynamics/interactions. Neither of the solutions is satisfactory for this dissertation's purpose.

As this method is desirable due to its familiarity to VR experts, an implementation has been lightly explored. Due to a limitation of the GHC (Haskell) compiler at the time of development, the callback method was not possible to implement. A necessity of this system is being able to make calls from the external world into the Haskell implemented system. As Haskell is a garbage collected system, a persistent handle to the objects have to be created in order keep our simulation environment from being deallocated. The mechanisms for this are present in the GHC compiler; unfortunately, their usage leads to memory access errors. Because of this, the pursuit of this method was suspended in favor of one of the other methods.

FRP-VR as a Lock-Step Process

The Lock-Step processes method is one of the two identified ways to loosely couple the run the FRP and VR systems. As the two systems are to some extent dependent on each other, they have to be synchronized in some way. The dependencies can be reduced to an information exchange. The approach presented here follows the basic VR system's design and is a natural fit to the classical VR system approach. As explained in Section 6.1.3, VR systems are most often directly coupled to the classical graphics update cycle. In this model, a once per "frame" update application call calculates any new data and modifies the scene. Figure 7.2 shows this process graphically. The lock-step process is also how the input from devices is handled in VR systems. Since the VR system operates via a "frame locked" application structure, that structure can be used in the implementation also. The FRP simulation call becomes just another of the application steps.

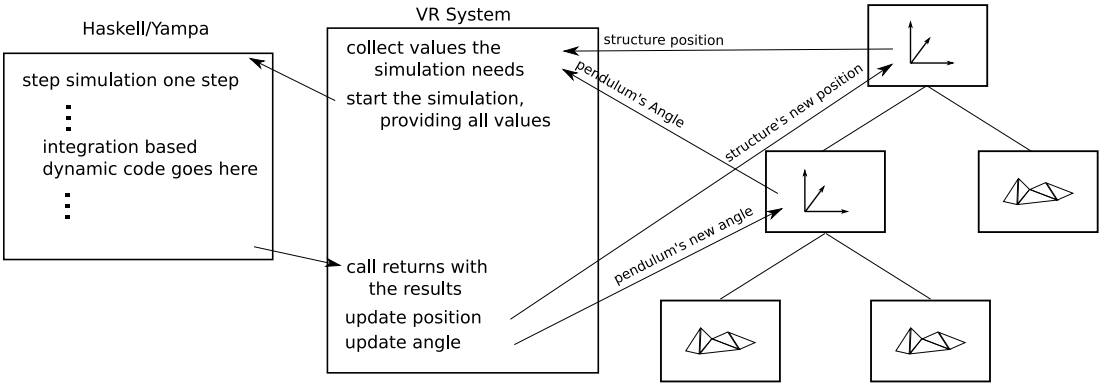


Figure 7.2: This diagram shows the implementation of FRP-VR using the lock-step synchronization, running the simulation only once per frame.

The systems can be exclusively run in step with each other, using this methodology. The basic process is described here using terminology used in systems like VR Juggler and OpenGL Performer. The process is: during the applications "preFrame" call, input data is transferred to the FRP implementation, it requests that the FRP to run the simulation, and finally transfers the information generated in the simulation to the appropriate place, e.g. the SG object transformations. Figures 7.2 illustrates the system architecture and the timing of the system.

This method also has its respective advantages and disadvantages. There are two main advantages to this implementation approach. The first is the match to the SG based VR system's design, since changes can only occur to the visual scene in sync with the frame (modifying the SG outside of specified times can lead to system crashes or untraceable further failures). The second is that the two systems only connect in one place, running sequentially. This simplifies greatly the synchronization of data (see the free-run method presented next for details). Another advantage over the scene graph coupled approach previous presented, handles all of the various possible dynamics in Chapter 4 the same.

The major drawback is that implementation does not implicitly make a connection between object semantics and the object representation, i.e. SG. This is not only a semantic difference, but also indicating that the user is responsible for proper delivery of the values to the right places on the VR side. One of two approaches here can be taken in this regard. The above presented idea leads to a single simulation that calculates all of the new behavior values, and, then, the user has to distribute the values. The other possibility is to have multiple small simulations, where each simulation represents a single object. This more closely reflects the coupled design above, but the assumption here is that the user is required to handle this instead of implicitly coupling the two systems at the system level.

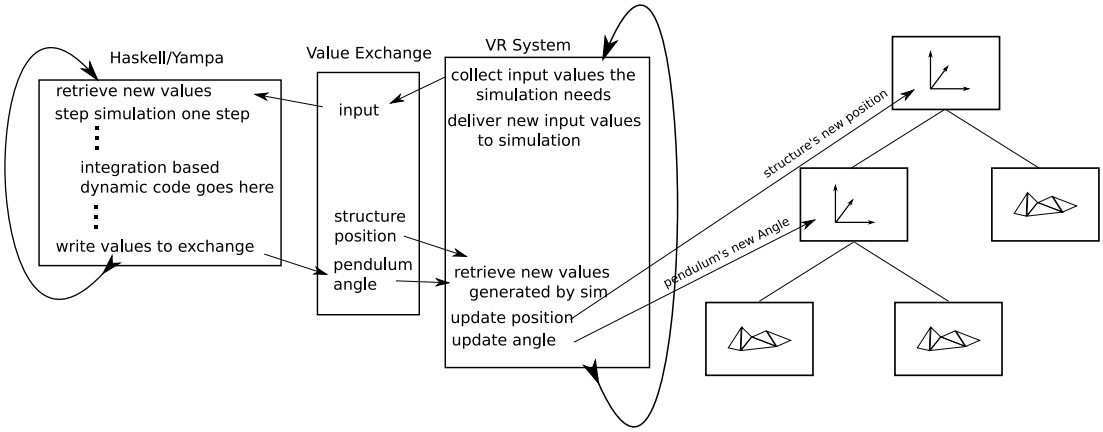


Figure 7.3: Free-wheeling threaded FRP-VR implementation.

FRP-VR as an Independent Thread

The frame-lock step implementation suffers from one substantial problem, one inherit to VR systems. VR systems are coupled to the graphics system, meaning the update rate is defined by that system. Simple linear functions behavior properly in most cases; however, for systems that are highly sensitive to the simulation fidelity, the relatively large time step of a graphics system can be a problem. Typically the best one can hope for in VR is a 60Hz update rate, and the rate can be expected to be above 15Hz. Most commonly, such sensitivity exhibits itself in the form of divergent behavior. For instance, in Section 9.2.2 a simple dynamic system that simulates a pendulum is presented. The system is defined by a second-order differential equation. Unfortunately, the system is unstable at a 60Hz update rate! This issue is not limited to simulations, but also prevalent in other VR sub-systems that require higher update rates. Examples include: input filtering (some devices offer up to 1000Hz updates), sound (approximately 44,000Hz), and haptics (minimum of 1000Hz).

A possible solution is to modify the lock-step strategy slightly to allow the two systems to run independently. This approach is taken for applications that require higher fidelity simulation and for sound and haptics. The implementation details of this change are limited to how the locking mechanism works between the Haskell thread and the VR system thread. The key to this method is maintaining data integrity between the systems. This is simply the classical multi-threaded system structure and is shown in Figure 7.3.

In building the multi-threaded system a few restrictions have to be considered. Interactive update rates need to be maintained in the visual system (as well as, any other modalities), while avoiding the introduction of lag on input processing. In other words, the system needs to be designed to meet soft real-time constraints on the visual thread. Naturally, the same can be said of the behaviors and interaction in the simulation.

They need to run at an appropriate speed. Taking other modalities into consideration makes this multi-threaded approach appealing. Exploiting this nature, the simulation can run as fast as possible and simply feed the different modalities current values as needed. Inputs can then be processed at a higher rate, potentially one close the actual input rate. High rate modalities like haptics can be feed with values at a higher rate, gaining fidelity in those systems. However, in designing systems with multiple update rate modalities, one needs to consider the validity of working with new (and therefore different) simulation values in different threads. Data consistency can quickly be a problem. Differences in the outputs across the modalities can be disturbing to the user experience.

7.2.3 Functional Reactive Virtual Reality

A design of our framework of support for DIVE creation, Functional Reactive Virtual Reality, can now be fully specified. The first step of the process is deciding on one of the proposed architectures for coupling the FRP system and VR systems. Since a primary requirement of the developed system is to remain independent of the VR system, the Scene Graph coupled design is undesirable. This leaves the two possible approaches, both of which can fulfill the requirements laid forth. Each has advantages and disadvantages. However, an implementation detail complicates the Lock-Step process. At the time of development, the Haskell compiler did not properly handle external calls into Haskell (still required memory was not deleted by the garbage collection system). This made it impossible for the FRP implementation to be called directly from the VR side. However, with a context of trying to create an easily understandable system, avoiding the thread synchronization issues of the independent thread design is desirable. Instead a solution that merged the two designs was developed.

A lock-step implementation was developed inside of a multi-threaded system. Although the systems are technically multi-threaded, enforcing the lock-step processing paradigm on this assures simplicity. The difficult system details to make this work remain hidden from the FRVR developers. This design has the advantage that any future movement to the independent thread method is only an incremental change. These changes are mostly limited to the synchronization software implementation and the conceptual work of how such a system should be held in sync properly.

The Yampa implementation of FRP and the VR system of choice are independent components. The independent system based design introduces the necessity of a solution for the exchange of information between the systems. The information exchange issue is compounded by the fact that Haskell and C++ have different data types and values must be marshalled between the languages. An additional issue that occurs with Yampa and VR systems is that both systems are designed to be in control. The design also has to deal with this control problem in some way.

The basic system implementation hinges on bridging the information flow between the two systems. To facilitate the exchange of information between the two systems, we have implemented a shared-memory with tagged values. Shared memory implementa-

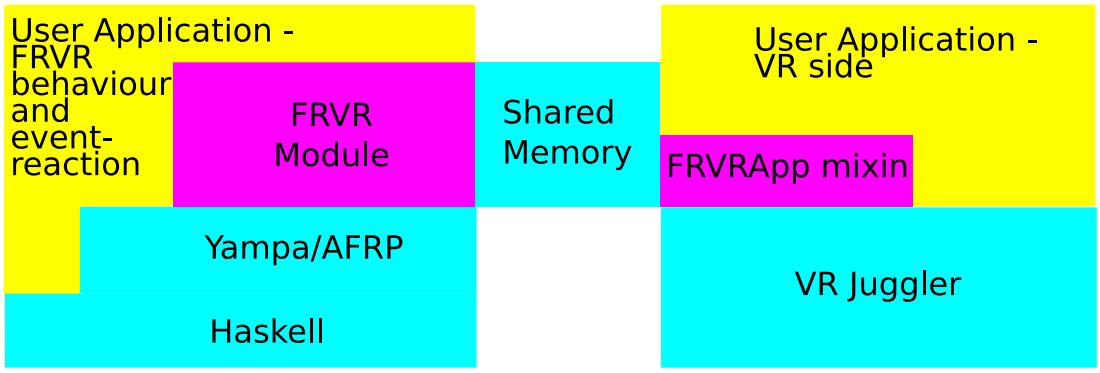


Figure 7.4: This diagram illustrates the basic FRVR system structure of a VR Juggler based implementation.

tions are indicative of many concurrent (multi-thread) systems. However, here we have followed a route that is used in robotics, where values are tagged with a name to allow easier access. This removes the standard issue of having to know where the appropriate memory place is, making implementation easier understandable and more flexible for the end user. This approach was chosen for several reasons and will be explained more in Section 7.3.2.

Beyond being composed of two separate systems and the shared-memory systems, the software design of the FRVR system follows a componentized approach. On the VR side, the modular nature of the FRVR code is necessary for exchangeability of the underlying VR software. The information exchange system is also modular, so that it can be easily replaced in case of expanded future needs. On the Haskell/FRP side, the modular approach is solely to improve the flexibility and usability of the implementation. A goal of all modules is to provide easy to use structures that handle the system implementation details for the developer where possible.

The final issue of importance that impacts the total implementation is that of control of the system. Both the VR system and Yampa simulation kernel expect to be in control of the frame loop. Since the display of the graphics requires at least a soft real-time update rate, the VR system needs to be in control of the system. For this reason a method has to be found to control the Yampa simulation kernel from an external source allowing it to only take a single step per frame. The basic idea is to lock the execution in the FRP thread and wait for the execution of the VR side to complete.

The following section addresses each of the modules individually. The modules are presented in the order of per frame run execution. In Section 7.5 the complete work flow will be explained.

7.3 FRVR Implementation

This section presents the implementation details of the different modules of the FRVR system. These are the modules identified in the design of the previous section. The initial VR implementation in the basic libraries, VR Juggler, is presented first. Second, the data exchange library is developed. Third, the Haskell libraries that enable FRVR to work are presented. Finally, a second VR implementation is shown based on the higher-level system, AVANGO.

7.3.1 FRVR VR Juggler Libraries

The independent system architecture chosen limits the necessary implementation requirements within the VR system to a few points. The FRVR implementation in the VR Juggler libraries consists of a small library. Its primary purpose is to assists the developer in the handful of calls necessary for controlling the FRVR loop. Because VR Juggler can use any of six rendering systems (OpenSceneGraph(OSG), OpenGL Performer™, OpenSceneGraph(OpenSG), OpenGL, VTK, NVSG), the implemented system must abstract from the concrete scene graph choice as much as possible. The basic functionality of the FRVR libraries can be combined with any of those rendering systems. The other issue that must be addressed is the data exchange between the systems.

The VR Juggler libraries demonstrate some of the system independent nature of the FRVR design. Each of the six potential rendering engines has a different application class, from which the developer derives their application. The FRVR libraries define a light-weight class for each of these rendering engine specific classes. As the code is identical in all cases, a mixin class performs the actual functionalities involved in establishing the simulation environment. A derived class simply manages the calls to both the parent class and the mixin class. However, the actual moment of the call to the simulation and the data that is exchanged has to be handled by the developer. The moment of the simulation has to be set by the developer, because the input has to be processed and placed into the exchange system before the simulation is started. In both cases, only the developer can know what data needs to be exchanged and where it comes from/goes to.

The VR system controls the Haskell/Yampa thread, starting it during initialization. As the implementation is multi-threaded, the VR system is also in charge of setting up the data exchange system. The implemented exchange system is a shared memory system further described in Section 7.3.2, but must be initialized by the VR side before the Haskell thread is started. The control functionalities of FRVR on the VR side are:

- initializing the shared memory system,
- starting the Haskell/Yampa thread by calling an externally defined function that starts the Haskell/Yampa system (described in Section 7.3.3),

- providing a single function for the developer to call that runs the Haskell system each frame, and
- cleaning up the system and killing the Haskell thread properly at the end of the program.

The threading and thread management is performed using the VR system's support for threading, as most VR systems contain abstractions of this functionality. The "VR Juggler Portable Runtime" (VPR) library is part of the VR Juggler framework that is responsible for this part. This library presents a cross-platform wrapper on basic threading and synchronization primitives that are required. The FRVR implementation leverages this for spawning the Haskell thread, yielding the process when the other half should start, and waiting on events data exchange that indicate the current thread should restart.

Two interfaces to the data exchange system are provided on the VR side of FRVR. The data exchange libraries themselves are written in C++. Using this interface, the developer is responsible for much of the memory control, e.g. making sure things are properly allocated and coercing data from simple data arrays to higher level data types. A second interface is also available that addresses several shortcomings of this interface. The second interface is written in C. These calls assist the developer in the memory related activities. Moreover, the C functions mirror more closely the interface provided on the Haskell side. This benefits the usability of the system, particularly for those with less programming experience by removing the need to deal with the more complicated C++ interfaces.

The actual data that is exchanged could be anything the developer requires. Likely the user will be providing input to the simulation from VR Juggler's Gadgeteer input system and, after the simulation is complete, changing the environment based on the results. Because this cannot be known a priori, the developer needs to remain in control the timing of the call to the simulation during the application processing (the preFrame callback in a VR Juggler application). Generally, the user must place all needed information into the data exchange system and, then, initiate the FRP system to run. A single call to the preFrame function of the mixin class starts the simulation and blocks. After the return of that call, the simulation has completed. Then, the programmer needs to retrieve the updated values and deliver them to the proper place in the VR system.

7.3.2 FRVR Data Exchange Libraries

The connection of a VR system and the Yampa FRP system requires an interface to handle the exchange of information between them. The interface also needs to handle the fact that the implementation languages of the systems are different, Haskell and C/C++. This is important, as the basic types of Haskell differ from those of C based languages. The system proposed and developed here is inspired by robotics research efforts. There, such systems have been used to aid development, improve flexibility,

and increase system reuse by decoupling sub-systems. The interface design is generally referred to as a BlackBoard. However, they are only partially related to the more famous BlackBoard systems of multi-agent systems. A more complete look at BlackBoards can be found in [Cor03, Cor91].

A number of requirements can be laid out for the data exchange system. The previously mentioned ability to exchange between Haskell and a VR system is the primary requirement. Other requirements placed on the design include:

- add little processing overhead (speed, latency, etc),
- be replaceable with another system,
- support any type of information, but specifically typical CG mathematics,
- have little impact on usability, e.g. be simple for non-experts to use,
- provide a semantically based data interface rather than "CS" type interface (e.g. names instead of hashes or indexes).

Unfortunately, the authors have not been able to find any open-source freely available systems implementing the type of system required here. Network based multi-agent systems, such as Cougaar [HTW04] and the Open Agent Architecture (OAA) [MCM99], allow the desired system to be built, but are designed for much different usage and are over-sized. However, in some cases they may make sense to pursue. In Section 10.3.3, possible further work based on using such a system will be discussed. A minimal system that directly supports the needs of the FRVR system has been written. The interface is held close to traditional BlackBoard systems, so that the implementation is exchangeable in the future.

The implemented system is a simple shared memory, where all information is tagged, such that each side can find and insert data based on the named key. This concept is much simpler for users to use than more performance effective methods, as the user is only concerned with the information, not how it exchanged. This model follows that of the blackboard models used in robotics, where each service simply writes to the blackboard and all services can read, erase, and write values. In the implementation presently used, the tagging is done with strings. This decision allows higher flexibility for the user, but at the price of expensive string compares. For applications with extensive data exchange this could become an issue, requiring either a change to other indexing methods or specialized array values (e.g. for implementing particle systems).

The information exchange is limited to a few primitive types that address the needs of a computer graphics system. Beyond a few basic types, e.g. float or integer, standard graphics mathematics form the basis of the data support. The "Generic Math Template Language" (GMTL) is one of the few CG math libraries that is independent of a VR/SG system. It is also the library on which VR Juggler builds. As with most CG math libraries, it defines the basic set of types and operators that are needed for computer

graphics work. Those types are:

Vec a 3 or 4 valued Vector,

Point 3 or 4 valued representation, a Cartesian coordinate,

Quat Quaternion rotation/orientation representation,

Euler Euler Angle rotation/orientation representation,

AxisAngle an axis and rotation angle defining a rotation/orientation,

Matrix a 3x3 or 4x4 Matrix,

Coord a Point and one of the orientation types.

An additional type that is necessary for FRVR is an Event type. As discussed in Section 3.1.3, events in FRVR are specified to represent a possible occurrence. In the system developed this is directly modelled. The event type is a simple boolean value, though it is only asserted when present. The absence of an event with the given tag indicates that the event has not occurred. Four functions are provided for working with events. Functions to insert an occurrence and to remove an occurrence as well as a test for occurrence are the basis of the system. The final function checks for an event occurrence, but removes the occurrence before reporting its presence.

The exchange system, because of its position between the FRP system and VR system, has an additional duty. As the communication route between the threads, it has to facilitate the lock-step method. This is done through providing support for the necessary locking mechanisms. The event type combined with basic threading support can provide the required support. Special functions that wait for specific event occurrences are supplied. The event is present or not present, allowing this to be used as the semaphore in the system. In order to reduce the processing load, threads can be backgrounded and woken on event insertion. When the correct event for the semaphore is introduced they wake up and continue processing. To do this, the threading system has to be brought into the exchange system. This creates the unfortunate necessity of using the threading libraries of the VR system (or those compatible with them, e.g. the pthreads libraries). This essentially means that special versions of the data exchange system have to be prepared for each VR system. At an additional cost on both sides, this dependency could be removed. For the purposes of the dissertation, it was decided this was acceptable, since the system was designed to be easily replaceable.

7.3.3 FRVR FRP Libraries

The Functional Reactive side of the FRVR system is implemented in the Yampa system. The Functional Reactive Programming paradigm is presented in more depth and with all the necessary context information in Appendix A. Here, the portions of the libraries necessary to enable the FRVR design to use the Yampa system are presented.

The discussion will largely be kept at a general level that should not require in depth knowledge of the Yampa system or the Haskell language. Still some aspects of the discussion require a basic understanding of the functional programming approach, such as what pure functional means.

For the FRP side of the system, three aspects have to be addressed. Foremost of these is the design of the method in which the control of the system can be moved from the Yampa system to the external VR loop. A number of simulation control factors that are integral to the complete system are discussed in relation to the control displacement. Following that discussion, the developed interface to the data exchange system is presented. Finally, a specialized mathematics module that was developed for the computer graphics type of systems is introduced.

FRP Simulation Control

One of the major aspects of integrating the Yampa FRP system into the FRVR system is finding appropriate methods for the control of the complete system. As discussed in Section 7.2.2, various approaches to the control of the system could be approached and the lock-step approach with the VR system in control was chosen. As the Yampa system was designed explicitly to be in control of the simulation itself, methods to allow the control to be moved to the VR system had to be developed. A number of related control issues surround this discussion and are brought up here.

The Yampa simulation is, in a sense, an embedded environment inside of Haskell. The environment is built out of a single compound Arrow that contains the complete simulation environment. The simulation is specified in a single function call placed in a Haskell main function. This "run" function defines a simulation loop and returns only after the simulation is done. The Yampa simulation loop works on three basic steps, each of which can be thought of a lambda function callback. These phases are visualized in Figure 7.5. The first step is the "sense" function. This is a Haskell IO function, which detects new values as inputs to the system. IO functions are special functions that are allowed to access the outside world. The sense function is also responsible for providing the time delta for the simulation step. The second step is the actual running of the defined simulation environment. This is the actual Yampa simulation, which is designed to be a "pure" system, i.e. one not having side effects. The final of step is the "actuate" function. This function receives the output of the system and is also an IO function. It is responsible for delivering the calculated values to the proper place.

In order to give control to the VR system, the sense function can be used. Locking the Yampa loop for the lock-step strategy is best done here. As the new information input from the VR system will first be available directly before the lock is released, the locking needs to occur at the beginning of the loop. Unfortunately, implementing the locking solely in Haskell is not feasible. As the sense function is an IO function, it can be used for external calls. This means that the same functions that were used in the VR and exchange systems can be used. Since this is always the same, an encapsulating function reduces the locking and threading overhead to a single call.

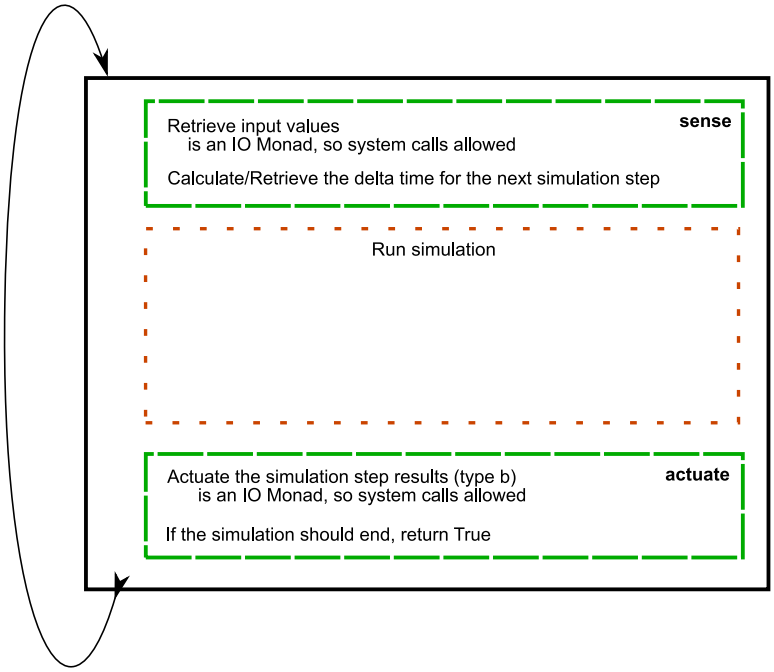


Figure 7.5: The Yampa Control Loop

The sense function also requires timing information. Unfortunately the existing Haskell time functions do not provide the resolution required. Calling standard system calls to get timing information of the required resolution would be possible, but places an unnecessary operating system dependency on the system. FRVR instead includes a Haskell wrapped call to the boost.time library [boo]. The boost time libraries have become a pseudo standard for getting high resolution time. They wrap time calls on most operating systems and including the calls needed to get to the level of CPU clock ticks, approximately microsecond resolution.

Typically this function is called at the end of the sense function, so that the timing is as close to the draw time as possible. Alternatively, it may be desirable to specify the time from the VR side, for instance of synchronization with other modules. By reading a time value from the data exchange and giving the appropriate time delta as a return this method can also be used. As the sense function has to be written by the developer, this choice is left up to them.

Having the developer writing the sense function brings another control issue to light. In order to make the programming of the Haskell side of FRP simpler, it would be nice to remove the need for the user to deal with the main Yampa loop and all of the Haskell IO programming present there. This can be done, but at the cost of a fundamental change to Yampa's design concept. Yampa is designed to be "pure," all input to and output from the simulation should be accomplished in the sense and actuate functions. It is possible to break the purity of Yampa, allowing the simulation

components to individually access and write to the data exchange. Functions of the IO type have to be used to make the external call. The importance of this is discussed in Appendix A.3.1. A way around this exists, by declaring the function "safe." This equates to assuring the compiler that the call will not crash the system. Performing such calls is considered bad style within the Haskell community. If we assume that our calls, presumably all to the data exchange system explained in the next subsection, are safe, then the main loop can be controlled by FRVR. The developer retrieves/writes all data as they need it and only the system control happens in the sense and actuate calls.

Keeping the purity of the system requires that all data retrieval occurs in the sense function, and all writing of results happens in the actuate function. The major disadvantage with this method is that all values that are needed have to be collect and given over as a large data structure. Each of the simulation components has to be provided with the correct values and the return values have to be similarly collected into a large structure. This often involves many values, making the data structure of values complicated. This packing and unpacking of values can become quite complicated and is pervasive to the entire coding of the simulation. All errors are found at compile time, but the error reports of the compiler can be somewhat difficult to follow with such large structures. This question of following the purity of Yampa or not is a more pervasive question, left to the programmer and a topic addressed again in Section 7.5.

Data Exchange Interface Module

The data exchange interface required for the Haskell side is, in some ways, more complicated than the C++ side. This is due to the way that Haskell integrates with the external world. Haskell has to access the exchange interface through the foreign function call interface provided. This places some restrictions on when and how the data can be accessed/written. Additionally, Haskell's data types are not identical to C++ data types, so special code to "marshall" the values must be included.

Haskell's foreign function call mechanisms provide methods to access C functions. This requires the development of C wrappers on the functionality of the exchange system. However, just wrapping the C++ calls are not enough, because of the marshalling issue. Haskell's built in marshalling functions can handle only the basic primitives floats, doubles, integers, etc and pointers to arrays of them. The developer will likely be predominately working with higher level math types discussed in Section 7.3.2. The C wrappers also need to be able to handle the type differences transparently for the developer.

The input into the Haskell marshalling system has to occur over one of two mechanisms. The simpler of these is having a pointer to a memory space that represents an array of a primitive type (float, double, etc). The more complicated version requires defining special handlers that understand the structure of the C typed memory to retrieve complex types. The basic types defined to support in Section 7.3.2 are limited

to types that can be represented as arrays of a single primitive type. The standard CG types almost all are represented as an array of floating point values in such systems.

Read requests from the Haskell side can be implemented in a straight forward way, since the data exchange system can provide a pointer to the internal array representation of all the different CG types. The flow of values from Haskell to the exchange system is a bit more challenging. Essentially, the Haskell write functions work by retrieving the C side memory chunk and writing an array of values into it. The question is what memory does it write to? Haskell provides allocation functions, but it is not guaranteed to be partitioned in as C's method. The implementation taken is the most certain and easiest for the Haskell side. Responsibility for the memory allocation/deletion is given to the data exchange system. In particular, the C wrapper is responsible for the memory allocation, conversion to the actual C++ data types, and the memory clean up.

The resultant interface on the Haskell side is a series of foreign function calls, retrieving and writing the various types. These calls are of the IO Monad type, which are considered unsafe calls. If the data exchange is only accessed in the "sense" and "actuate" functions, then there is no issue since they are IO functions anyway. However, if the programmer desires to access the data inside of the simulation, it could create a problem. If the developer chooses to take this approach, special wrapped functions that denote the calls as "safe" are provided.

Math Module

Haskell has a mathematically oriented syntax and even supports mathematical entities, like sets and Arrows. However, Haskell does not include all of the mathematical types and operations that are important for computer graphics. Given that a few computer graphics modules exist in Haskell, it is surprising that the basic types, such as matrix, vector, quaternion, are missing. The one type that Yampa has is the AFRPVector type class, as part of an AFRPVectorSpace. The type is actually used for the internal stream implementation (see the more in-depth discussion of this in the FRP coverage in Appendix A) and does not define such basic functionalities as the dot product. A computer graphics based Math library was developed to overcome this limitation. The library developed can be separated out as a stand-alone module, making it usable in other contexts.

In order to assure completeness and provide a familiar interface to users, a library based on the Generic Math Template Language (GMTL) was used. The Math module has been written with type classes and operations matching those of the GMTL module as much as possible, though concessions have had to be made. Mostly this is due to the way in which conversion of types is handled in Haskell. In essence, extra conversion functions had to be defined. This leads to a situation that requires the developer to convert to meta-types and then back to concrete types (although in reality only one change to the data is performed). An example of this is converting from an AxisAngle

to a meta Rotation type, which can then be changed to a quaternion. The actual code appears as:

```
(toQuat (AxisAngleRot (AxisAngle (Vector3 0 0 1) 90)))
```

Listing 7.1: AxisAngle to Quaternion Conversion

Conversion over this method add an extra function, but is a similar behavior to that present in GMTL. Several Haskell research projects have the potential to provide methods to eliminate the need for such methods in the future.

7.3.4 FRVR AVANGO libraries

The prior sections presented a complete FRVR system, spanning the VRJuggler system, the data exchange, and the FRP libraries. In this subsection, the implementation of FRVR in a second VR system is developed. The implementation demonstrates the system independence of the implementation and also an example of FRVR integration in a more complex system. The AVANGO VR system was chosen for the second implementation. AVANGO is described in Section 6.2.1 and in the papers [Tra03, Tra99] (AVANGO was formally referred to as avocado).

The AVANGO FRVR implementation is presented in three parts. The first sub-section introduces the method used to integrate FRVR into AVANGO, including introducing AVANGO details as required. The remaining two parts present extensions to AVANGO types that ease the use of FRVR. The second sub-section presents an extension to the AVANGO system that permits the values coming from the FRP side to be inserted into the data-flow system without requiring programming in a low-level language. The last sub-section presents an analogous extension that allows programming at a high-level for insertion of data from the data-flow system into the data-exchange system. Portions of this work were presented in a GI VRAR Workshop paper [BB07b].

FRVR-AVANGO Implementation

The implementation of FRVR in a higher-level VR system requires a different approach than that taken with a basic VR system. Since the higher level system is already composed of methods meant to encapsulate and specify higher level (beyond that of spatial arrangement of graphics primitives), the integration of FRVR needs to complement those structures and work within the spirit of those systems. In the case of AVANGO, the higher-level system is a data flow based approach. Another issue in many higher-level systems is the control structure. For instance, in the case of AVANGO, access to the control loop is closed to the developer. This forces solutions, which work within the parameters of the system.

The basic integration method adheres to AVANGO design principles and is shown in Figure 7.6. The access to the data exchange and controlling structures required are encapsulated into a special Service, *fpFRVRService*. The *fpFRVRService* initializes and maintains the connection with the Haskell thread and handles movement of data into

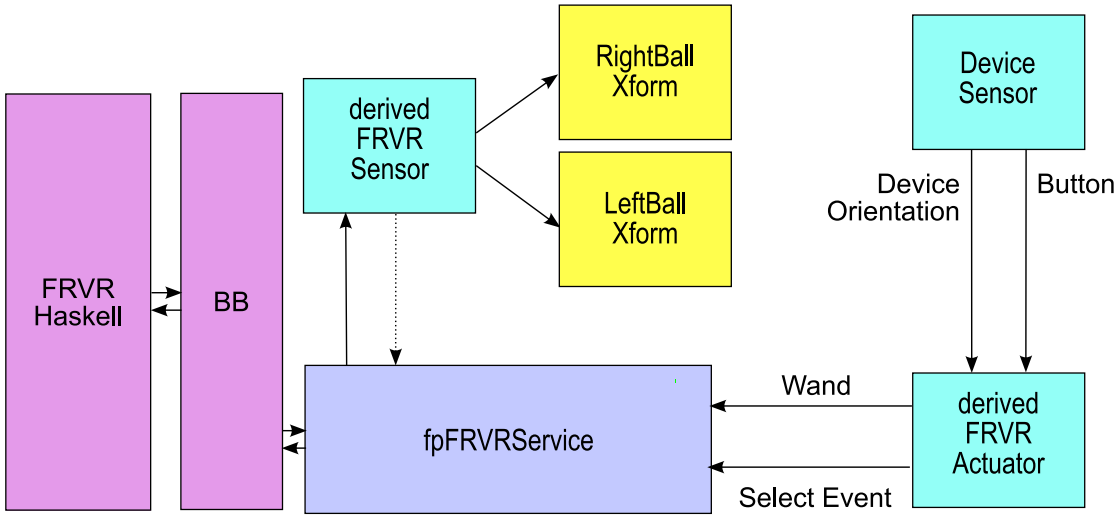


Figure 7.6: This figure shows the flow of information in the FRVR AVANGO coupling, on hand an implementation of a Newton's Cradle. The derived Sensor and Actuator classes are extended with Fields to deliver/receive the values from/to the FRVR system.

and out of the BlackBoard. The service is registered with AVANGOs control structures. Through this, it is included in the AVANGO update cycle and is called automatically every frame.

A special class of Sensor, *fpFRVRSensorBase*, implements the basics of connecting to the *fpFRVRService* to retrieve values. Sensors are the part of the AVANGO system responsible for the input of external information and are called after the services are run. Sensors are part of the data flow graph, meaning they have special output fields that inject information into the graph. Values from the data exchange are inserted into the graph by the Sensor. The Sensors inform the *fpFRVRService* of values of interest, using their names and initializing the values. Thereafter, the Sensors retrieve values every frame. The Sensor's implementation packs the values into Fields, thereby inserting them into the dataflow graph. The actual handling of the values and distribution into the Fields must be handled by the developer. An alternative Sensor implementation that can be used without extensive C++ knowledge is presented in the next part of this section.

The flow of information inside the FRVR AVANGO implementation follows as illustrated in Figure 7.6. The Haskell FRP system is executed by the Service every frame, before the Sensors update. The Haskell FRP simulation side updates all values, including any asserted Events, as required. The Sensors, which access the *fpFRVRService*, then retrieve the new values and set them into the data flow. When the FRVR system is used to control dynamics in the environment, the setup is complete. The values can be routed via the Field mechanisms to the proper place, using scheme to perform the setup.

Incorporating interaction into the system is possible using similar techniques. Interaction, however, is a much more complex design question than simply driving dynamics with FRVR. One problem is that AVANGO has a number of different interaction mechanisms. Here, basic support for interaction using FRVR is developed, completely bypassing the AVANGO methods. Interaction controlled in FRVR relies mostly on the correct input values being placed into the data exchange. From there, the Haskell FRP code retrieves the values and events from the VR system, processing them as appropriate. The *fpFRVRActuator* class mirrors the *fpFRVRSensor* and is responsible for inserting data in the data exchange. In other words: *fpFRVRSensor* reads values from the data exchange system and *fpFRVRActuator* writes values to the data exchange system.

An example of how FRVR integrates into AVANGO helps illustrated the method. Here, we demonstrate the connectivity for a dynamic, interactive ball that is part of a Newton's Cradle - the simulation code for this example is described in the Examples Chapter in Section 9.2.4. A flow diagram of is provided in Figure 7.6. The two outermost balls are independently modelled portions of the orthogonal scene graph. A Sensor retrieves the orientation of the RightBall and LeftBall. The Sensor has fields named as such and FieldConnections to the transformation nodes automatically deliver values every update (which occur every frame). To include interaction, an Actuator with a Wand matrix field and an Event field for selection, inserts information into the FRP system. The Wand and Selection fields are then connected via FieldConnections to the input devices, such as the stylus and stylus button.

Run-Time Reflective Fields and Field Connections

The design described above required the developer to either implement a derived Sensor type or embed the calls to the *fpFRVRService* into their own class. As one of FRVR's design goals is to simplify the programming of interactive dynamics, requiring the use of AVANGO's complex C++ class inheritance is less than optimal. In this section, we describe an extension to AVANGO that enables a style of reflection on run-time defined Fields.

This extension leverages the fact that the shared memory of FRVR uses a named memory approach typical of BlackBoard architectures for the exchange of information between the VR system and Haskell side. By maintaining two lists, one containing the names of values to be retrieved and the other holding the actual values, one can dynamically insert the FRVR generated information into the AVANGO dataflow system. A sensor receiving values, all of a single type, can use the Multi-Field version of AVANGO's Fields. Unfortunately, this still requires programming by the user, as no existing AVANGO nodes are prepared to take Multi-Field values and extract a single value out of the Multi-Field.

In order to make the multi-field approach useful, AVANGO's FieldConnection principle is extended by placing a connector in between. A special connector class, *fpNamedFieldConnection*, was developed that attaches to a Sensor that has such a

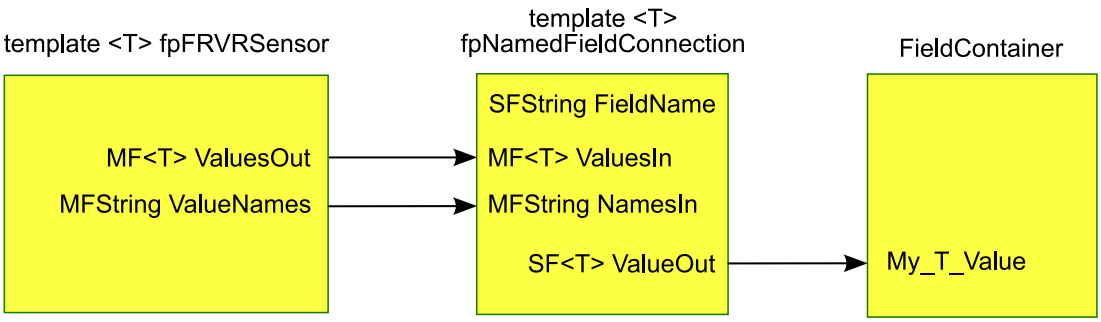


Figure 7.7: The class descriptions of MultiField reflection system of FRVR-AVANGO. The *fpFRVRSensor* and *fpNamedFieldConnection* template classes allow access to the run-time defined "Fields" as if they were normal Fields through their tags.

multi-field output. The connector is set to sort out a specifically named value from the list. The output of the *fpNamedFieldConnection* is simply a single value Field of the same type. Figure 7.7 shows the connector is used in the dataflow system. In order to avoid a possible frame delay, caused by the Field update mechanism, the *fpNamedFieldConnection* processes the input on any change of the input Multi-Field. The processing of values from FRVR, through the extended *fpFRVRSensor* and using *fpNamedFieldConnection*, can be seen in Figure 7.8. In order to make this a more agile design pattern, both *fpFRVRSensor* and *fpNamedFieldConnection* are implemented as C++ templates, parameterized across the Field/data type.

Named Tuple Connector

The Actuator can also be parameterized to assist the user. However, the Actuator suffers from a slightly different problem than the Sensor. In this case incoming information is a tuple, the string tag and the actual value. Unfortunately, AVANGO does not have an available support mechanism for this. Simply having two Fields, for the name and value, does not solve the problem. This is because of the way that AVANGO's update mechanism works. A special asynchronous call is executed when a Field is updated. When the node containing the Fields updates in the data flow cycle a synchronous function updates the node internals based on the Fields. There is no way for the node put together two asynchronous Field updates, given that there must be a provision for any number of different values to be given. A related issue is that there is not a mechanism to provide a Field that contains the name that updates at the right time (there is a way to force it to update every frame, but that leads to unnecessary copying of expensive strings).

The data required is in essence a simple tuple and should really be handled this way. Unfortunately, there is no mechanism to handle them. The solution developed uses a Field parameterized on a std::pair, templated to the correct value type. To address the issue of where the tag comes from, we again developed a new Field Connection

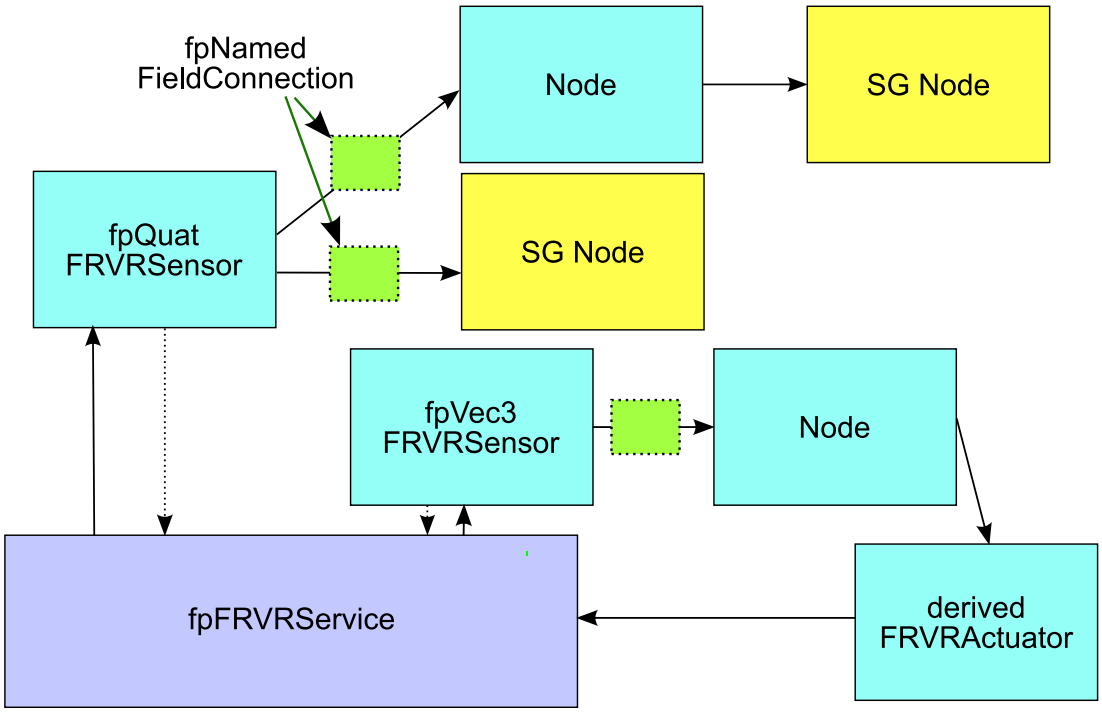


Figure 7.8: The flow of data through the extended FRVR-AVANGO system. Incorporating the fpNamedFieldConnection objects, removes the need for extending C++ classes to access FRVR values.

mechanism, *fpPairingFieldConnection*. The *fpPairingFieldConnection* has two inputs and a single output, as seen in Figure 7.9. The inputs correlate to the value and the tag. The name of the value typically does not change after creation and can simply be set once upon establishing the object. A FieldConnection from the value to be placed in the data exchange to the *fpPairingFieldConnection* Value is created. The output of the connector is the paired value, which is connected to the *fpFRVRActuator* via FieldConnection.

7.4 System Flow

An inspection of the flow of data through the FRVR system is provided in this section. Not only does this provide a good overview level view of how FRVR operates, but it also is informative for the program developer, particularly for the non-VR expert. Understanding how the system flow works, can impact heavily the developed code structure, inform decisions about data ownership, and even how a program is developed. In this section, the flow will be presented. Most of the discussion of its impact is delegated to the following section on the expected developer work flow.

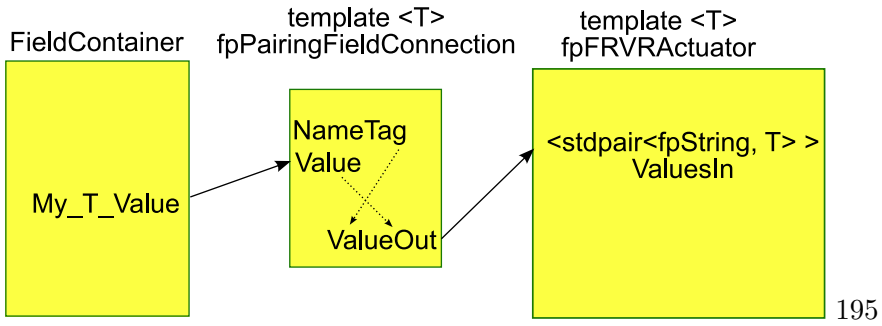


Figure 7.9: A helper for Field Connections requiring a tuple Field. This variation is specifically for the fpFRVRActuator taking a name as the first value.

The workings of FRVR occur as part of the VR loop that was explained in Section 6.1.3. All of FRVR occurs inside of the application processing phase. The presented information here assumes a simple VR system, though all systems share the same general flow principles at their core. The diagram in Figure 7.10 of FRVR's flow highlights the method used. FRVR's process is essentially a three phase process: gather and transfer of input data, simulation, transfer and distribute simulation results to output systems.

The start of FRVR's work cycle coincides with the start of the standard application processing cycle. Assuming that interaction is being processed in the FRVR system, the first steps of the FRVR cycle is the retrieval of input data. FRVR potentially requires two types of data, the actual user input and inputs that are from virtual components not directly controlled by the FRP side. Typically, VR systems provide tools that automatically retrieve the user input via hardware abstraction layers. This is usually performed in a step just prior to the application phase in VR systems. If these mechanisms are being used, then the values are simply routed to the data exchange. Otherwise the device polling can be considered part of the FRVR cycle. The next substep is to transfer any "virtual" information that resides in the VR side of the system that is required.

After all input data for the simulation is present in the data exchange, the FRP simulation side is signaled to start. At this point, the VR thread enters a blocking function awaiting the FRP side to finish the simulation. The VR thread is put in a sleep modus to reduce unneeded processing. The FRP thread is woken at this point, via the release of a semaphore held in the data exchange. The FRP side locks the processing and starts the second phase of FRVR processing.

The simulation phase starts with the retrieval of the input values and preparing them for use in the FRP simulation. This is performed in Haskell, in the sense callback function. The main work here is preparing the proper tuple of values to hand over to the simulation kernel. Depending on the input approach taken (refer back to Section 7.3.3 for a discussion of this), only the time delta has to be retrieved. At that point the Yampa kernel is run, executing the provided simulation environment. New values based

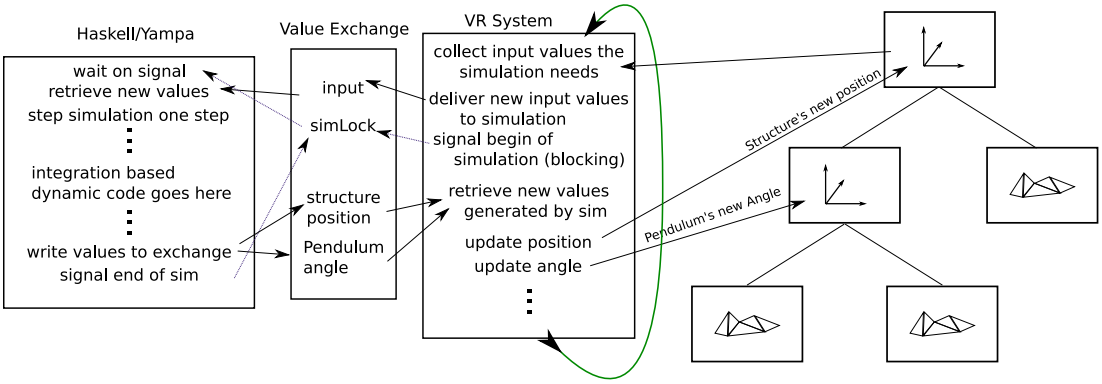


Figure 7.10: The timing of the FRVR system developed is demonstrated and shows the flow of data through system. The example case here mirrors the example used in the system architecture development of the previous section.

on inputs, time, and possibly internal values are created. The values are threaded out of the Yampa simulation as return values. The actuate function then writes the output values into the data exchange. In most cases, the new value simply overwrites the old value. Finally, the Yampa/Haskell signals the end of simulation and blocks waiting the next simulation step. Again, the thread sleeps to avoid unnecessary processing.

The third phase of the FRVR is the processing of the simulation results on the VR side. This starts with the VR thread waking up and locking control of the processes. The VR side then retrieves the values from the simulation and updates the appropriate places in the output systems, for instance positions and orientations of scene graph components. Any further processing that the VR side needs to do before the draw is performed. Finally, the actual render call is initialized, handing over control once again to the VR system.

7.5 Developer Work Flow

Given the multi-system environment of the FRVR solution, it is informative to look at the expected work flow of the developer. FRVR spans two programming languages, plus a third embedded language FRP, and deals with at least three different systems (FRP, VR System, data exchange), often with the addition of the scene graph. Insight into how development is expected to proceed, sheds light on all the components of the developer's process.

The first step for the developer is deciding what things are to be dynamic and/or interactive in the environment. The first part of this dissertation showed that the potential DIVE components are many and varied. This step is important, as the developer should consider firstly what is desired and then how best to implement it. For instance, adding dynamics to sub-component need to be thought out well in advance, as it can heavily impact many parts of its development.

Generally, upon deciding what will be dynamic, the computer graphics side of the matter will be taken care of first. By this, it is assumed that details like the modelling will be at least planned, as the structure of the model can have a large impact on the way the dynamics have to be produced. For instance, for a moving object, the object movement as a whole has to be determined as well as how components will move. In a scene graph setting this entails developing the hierarchy that the model follows. This, though, is standard to all computer graphics applications that include dynamics or interaction.

After that is accomplished, the data exchange interface, i.e. what data will be flowing to and from the FRP side of the system should be determined. This is essentially a task of defining the interface between the systems. For standard spatial behaviors, this is simply a coordinate and orientation type. For more abstract things like autonomous entities, this may be a goal position or even values related to the current actors mental and emotion states.

Armed with the interface and how those values will affect the model or dynamic values that are changing, the developer can plan the simulation implementation. While this smacks of typical rhetoric, in the case of Yampa programming planning this has proven to be a vital step. Because of a number of factors related to the language being used and the way that Yampa works, the typical "hacking" method of programming used in the VR community is difficult. The pure nature of Yampa means that even slight changes to the information being passed around can mean changes to many functions (as the values have to be threaded through the entire process in a pure setting). This threading process is difficult and tedious when the values are changed often. The developer is much better served in fully developing the dynamics and interaction that must take place at a conceptual level first. Then interfaces and the values that will be changed should be developed. After this, algorithms implementable in Yampa/Haskell can be developed.

As mentioned in the introduction to Arrows in Appendix A.3.2, the Arrows represent computations instead of functions in typical Haskell. This has the advantage that computations - which can be thought of black box processes - can be simply strung together. This is precisely what the Arrow function $\ggg$ does. Using this nature the program is easy to develop at a conceptual level, using methods like data flow diagrams. The authors, particularly during the learning phase, found this to be an indispensable approach to Yampa programming. After developing a basic idea of how the dynamic and interaction could be done, a flow type diagram of the functionality was developed. This permitted a good understanding of the design and also showed problems in the algorithm developed. The flow diagram is easily transferred into an actual implementation as it directly models the method used with Arrows. A method of support base on this diagram approach is present in Section 8.5.2.

A critical consideration for the developer using FRVR is the issue of "ownership" of values. Because there are two different systems running that need access to the same values, clarity as to where the definitive value is going to be kept is important.

If the user is interested in maintaining the purity of FRP, it is clear that all values should "live" in the VR side. However, if the simulation specifies the dynamics and even interactive dynamics completely, this does not always make sense. Although the simulation is in complete control of the dynamics, it is not in control of the value. Worse, this means that the value is transferred twice, VR → FRP → VR, where only the FRP to VR transfer is truly necessary. The implementation used requires enough calculations to make this a worthwhile consideration. Examples of both approaches will be provided in the examples of Chapter 9.

7.6 Discussion of Resultant System

This chapter has developed a system that supports the development of DIVEs in immersive environments. Requirements for a proposed a system for support were laid out, a software architecture was proposed, and a concrete implementation of that system was developed. This final section of the chapter, investigates the match of the implemented system to the collected requirements. The discussion of the system is divided into the same three sections as the requirements were presented in Section 7.1.2: dissertation definition derived, Software Engineering, and DIVE requirements.

7.6.1 Dissertation Intent Derived Requirements

The requirements extracted from the definition of the dissertation listed in Section 7.1.2 are mostly qualitative in nature, making them difficult to verify. The non-monolithic system and building on existing VR systems requirements are also part of the software engineering requirements and are discussed in the next sub-section. Four requirements remain to be addressed here. Two requirements address the time representation. One requirement deals with the representation used matching the user's perception of the dynamics and interactions. The final remaining component addresses the coding method in relation to usability.

The requirements defined that the user should not have to deal with time deltas and that the time representation should address the hybrid nature of the user's perception of the world. The FRP concept is precisely designed to address this hybrid nature. Yampa still requires the user to deal with time deltas, explicitly feeding them to the simulation kernel. However, the user does not have to calculate their dynamics using the time delta, but instead only specifies the continuous nature. The system deals with the time delta transparently. In certain cases, such as those discussed in Section 7.3.3, the developer may require working with the time delta. This is for functionalities involving systems external to the simulation environment. Flexibility to enable such time delta manipulation externally is included as probable need exists, at the cost of minimal exposure of the time delta to the developer.

We have hypothesized, supported partially by the time discussions of Chapter 3, that the human perceives their world as a type of hybrid system. This was the basis

for the hybrid requirement just discussed. Matching this conjecture with the system of support lends support to the question of whether the system matches the developer's perception of the world. However, it doesn't fully address the requirement. Elliot proposed the FRP concept specifically to better match the human perception of what was happening [EH97]. The question remains whether people really perceive dynamics as mathematical, forward kinematic, continuous functions. However, it is probably doubtful that for non-physically based objects the user truly thinks of the dynamic as a mathematical function. Naturally, a study of how the user actually perceives the dynamics would be best, but no such studies are known to the authors and performing such a study is outside of the scope of the work. As the FRP based system seems to be at least an incremental step further than other possibilities, the requirement will be considered partially fulfilled. The only other system that dealt with continuous time was the constraint based methods, but they require "backwards" specification of the dynamics and are likely to be more difficult for the average developer. In the next Chapter, this shortcoming is addressed.

The final requirement that comes of the project motivation and definition is to improve the usability of the system, particularly for non-VR experts. Although, no tests have been performed to proof or disprove this, we feel that the Yampa system makes some strides at improving usability. Foremost is that Haskell and Yampa are based on mathematical syntax and style of programming. For the targeted groups of people, scientists and particularly engineers, this is an advantage. Engineers and scientists are often used to dealing with integrals for time based functions. In addition, we feel that the way that Yampa is programmed is beneficial. Because of several factors — threading of values, recursion, pure functions, etc. — effectively writing Yampa based code requires considering the algorithm of what is to be done before programming. Forcing a conceptual understanding of solving the problem before programming is a benefit to all, though sometimes frustrating for traditional "hackers." This process does not traditional software engineering approaches like iterative design, but rather impacts the implementation of each iteration. For instance, the examples of Chapter 9 effectively show an iterative development from a very simple system to a more complex system. The ease of programming is addressed again in Chapter 8.

7.6.2 Software Engineering Requirements

An analysis of the software design reduces to the question of whether the software architecture and the resultant implementation match the software engineering requirements laid out in Section 7.1.2. The requirements of having a non-monolithic design and cross platform and VR system support are met and demonstrated two-fold. The VR Juggler implementation allows connection with at least six different rendering engines and is further expandable as new rendering engines are incorporated in that project. As the both the VR Juggler and Haskell/Yampa projects are cross platform in nature the design also address this capability. The second implementation in the AVANGO VR system demonstrates the ability of the FRVR design to be incorporated in various VR systems, including those with existing higher level functionalities.

The chosen system for support of DIVE functionality, FRP, is by nature well suited to fulfilling a number of the software engineering requirements. The FRP concept and the Yampa implementation are specifically designed to facilitate code reuse. A shortcoming of the system, when following the pureness of Yampa's design, is that the packing of input and output can create difficulties for direct reuse. The programmer may have to rewrite the packing and threading portions of the code to make it work. An alternative, non-pure usage of Yampa was described in Section 7.3.3. Following this method, many of the difficulties related to routing values can be avoided, but at the cost of having non-pure implementation. This can cause debugging difficulties and also make verification of the code more difficult. Verification of the code using the pure approach of Yampa is possible using formal (mathematical) proofs (see [Cou04] for details). Various soft real-time applications of Yampa have shown that it can meet that requirement [CNP03, Hua96, PHE99].

Yampa provides a framework for the developer to create dynamics, interaction, and event based changes. The developer is expected to write the actual functionality based on the basics provided. It is to be expected that the developer can create further functionalities through extension. However, Yampa's design hides the implementation from the user. This means that any extensions that require the internal workings of Yampa cannot be done without modifying Yampa itself. This was done in Yampa for specific reasons, mostly because access to the internals opens the possibilities that the user could reintroduce the "space-time" leaks - one of the issues Yampa was designed to address (see Appendix A and [Cou04] for details of this). The extensibility requirement for the system is, therefore, only partially fulfilled. The topic of extensibility is addressed again in Chapter 8.

Scalability of the design is an interesting point. The design and a few factors create good potential for scaling up as the DIVEs becomes more complex. The design involving the BlackBoard like system is advantageous for scalability on a high level. Since the data exchange system built was designed to be replaced in the future, it is conceivable to switch out the implemented shard memory based system with a full multi-agent based Blackboard system, e.g. Cougaar [HTW04] or OpenAgent [MCM99] architectures. These systems are explicitly good at scaling up. They would allow multiple Yampa simulation kernels to run, potentially on multiple machines. Since most of the dynamics and interactions identified are independent of each other, a distributed computational view can be built on having individual dynamics/interaction distributed.

A second possibility that is even easier to implement is using the native abilities of Haskell. Several libraries that enable concurrent processing exist and provisions exist for FRP to be used as such. This is essentially applied via the compiler, without modification. The utility of this is dependent on the number of cores on the computer running it and number of concurrent system processes running (e.g. does the VR system have other threads running at that time).

Finally, all components of FRVR are either open source projects licenses that are compatible or are developments of the author. This means that the complete project

can be released as open source, fulfilling the last of the software requirements listed. FRVR was released as an open source project in March of 2008. The developed code is released under the LGPL license and modified portions of the Yampa system under the conditions of the original release.

7.6.3 DIVE Requirements

The requirements related to the creation of DIVEs are large enough that they will not be address singularly here. Many of the requirements for DIVEs are fulfilled through the choice of the FRP paradigm as the basis system. Select other functionalities are won through the design of FRVR. However, it is shown that a number of functionalities are still missing from the system.

The FRP system in itself provides methods to fulfill a number of the most basic requirements. The most critical requirements of different time representations are natively fulfilled by the Yampa system. The event type of Yampa matches the type that is needed for expressing system of change, like those required for Interactive Dynamics. Through the switching mechanism of FRP, run-time changing systems are easily created. The mathematical nature of Haskell makes expressing filters easy, as well as, dealing with sets of input that would be generated when receiving all input values.

The FRVR design adds in support for a few specific functionalities. The ability to perform interaction through the system is enabled through the bi-directional flow of data in the data exchange. Discrete events generated in the scene graph (e.g. collisions) and user input can be delivered to and processed in the FRP system. Continuous inputs can likewise be done. The only requirement that is not explicitly handled is receiving all user input values. This is mostly a matter of programming the ability on the VR side, as there are no restrictions to the exchange that would hinder it. The discussion of the system architecture in Section 7.2.2 includes ideas on an alternative method for solving this, retrieval of input at a higher frequency in the Haskell/FRP itself. Finally, the ability to change time and even to pause time are enabled through the developers freedom to specify the time delta that the FRP simulation uses. Time skewing can be done by simply scaling the inputted time delta. By setting the time delta to zero, the system is effectively paused, i.e. the dynamics do not move forward. This pausing functionality is discussed more in Section 8.3.3.

A number of required functionalities are not yet present in the FRVR system. The requirements that are not addressed are:

- ability to manipulate time, including the ability to
  - define multiple time "speeds," potentially per dynamic
  - undo, including over all time representations (where possible)
- enable handling of input
  - including all values, not only the most recent
- provide an ability to record and play back Dynamic Interactions
- provide an ability to undo Dynamic Interactions

- reaction to events by
  - continuing frozen dynamics - no time passed
  - continuing frozen dynamics - as if they had run
- in/out functions
- blending
- user defined transition functions
- exchangeable (interpolation) kernels
- basic sets of interpolation kernels (linear, sinus, etc)
- support for different timings of function transitions

These requirements need to be address in order to have a complete system. The next Chapter introduces extensions to the FRP system that will provide mechanisms for all of these requirements, excepting the requirement of handling all input values discussed previously.

Chapter 8

FRVR Extended Functional Reactive Programming

The Functional Reactive Virtual Reality system developed in the previous chapter meets many of the requirements laid out through the combination of the Yampa FRP system with existing VR systems. In the discussion of the resultant system, it was seen that a number of the requirements from Section 7.1.2 were unfulfilled.

This chapter presents the developments that address these unresolved requirements. The first section of the chapter revisits the requirements list and the results of the last chapter. The requirements that need yet to be addressed are collected and listed. In the remaining four sections, the requirements are addressed.

In Section 8.2 improvements to the existing FRP functionalities are presented. In Section 8.3 new extensions of the FRP functionalities are developed. Section 8.4 expands on those extensions with capabilities specifically to support Interactive Dynamics. The content of these three sections is technical in nature. A basic understanding of Haskell and an understanding of Yampa are required for many of the details. Appendix A introduces the necessary details of both Haskell and Yampa. Finally, Section 8.5 presents work that strives to reduce the programming difficulties inherent in the system.

8.1 Requirements

A discussion of the resultant system developed in the previous chapter can be found in Section 7.6. There, the FRVR system, as the combination of the Yampa FRP system, a VR system, and connecting developments, was compared with the requirements for the system laid forth at the beginning of that chapter. In that discussion, it was noted that many of the requirements, particularly from the DIVE support requirements, were not met by the existing framework. An inspection of the missing functionalities, shows

that they are limited to the FRP side of the FRVR system. An additional point that can be further addressed is the usability of the system.

In the remainder of this section, these requirements as well as a few additional functionalities that need to be addressed will be discussed in the context of their implementation in our FRVR system. These required developments can be broken up into three groups, based on how their implementations can be approached. The first set deals with functionalities that Yampa provides, but are unsatisfactory for the needs of VR and our system. The second category is functionalities that are ideas not yet present in the FRP implementations. In specific, these are extensions to the Yampa implementation. Finally, a small collection of important works that address the usability of the system are outlined.

8.1.1 Improvements to Yampa Components

Several aspects of the Yampa FRP system are unsatisfactory for the needs of the FRVR system and to meet the requirements laid out previously. The main point that needs addressed is maintaining a valid simulation within the restrictions of the real-time VR system. Although, this was never explicitly listed as a requirement, the dynamics generated need to be correct with respect to time. The basic issue is maintaining a valid simulation of the environment in the face of variable and even potentially large time deltas between frames.

The choice of the lock-step architecture for the overall system design couples the simulation to the update rate for the VR application. As mentioned in Section 6.1, the frame (scene update) rate of VR systems is often between 15Hz and 30Hz, though higher rates up to the refresh rate display system (approximately 60Hz) are of course possible. This means, the simulation can be expected to be running at roughly 15Hz and 60Hz or, in other terms, at time deltas between 0.067 and 0.0167 seconds. What is potentially more of a problem, is that VR systems are susceptible to random "hiccups," that is sudden slowdowns for a single or series of frames. The time between frames can drastically change through these, doubling or more. Because of the tight coupling of the update rates, the stability of the simulation comes into question.

As the simulation is based on integral calculus, the time step is of crucial importance for stability. Recall that numerical integration is valid on the assumption of a small time delta. The integral implementation of Yampa uses the rectangle (also known as the midpoint) rule, meaning it is particularly dependent on this assumption. A discussion of numerical integration methods for dynamics can be found in [Par02].

Unfortunately, Yampa was designed on the assumption of a well behavior (actually, on a developer specified) time delta. The choice of the rectangle rule for the integration method is an indicator of this (Courtney discusses this choice in [Cou04], where it was determined to be "good enough"). Although changing the numerical integrator would be a possible approach, it does not solve all issues, such as the large steps on "hiccups." The basic solution we propose is the use of sub-sampling techniques.

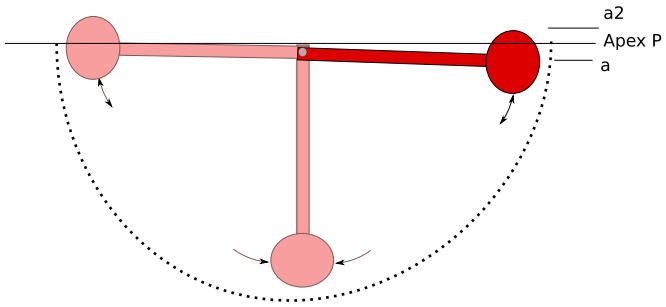


Figure 8.1: The diagram highlights a portion of the sampling issues FRVR. Here, a problematic case involving the dynamics of a pendulum is shown.

Using these techniques each time delta is potentially sampled (the simulation is run) multiple times, effectively reducing the time delta for each step.

Different implementation approaches could be used for sub-sampling. The simplest conceptually is just to set a minimum step size and sample the simulation at the highest level as required. However, it is important to note that the resolution required is different for different simulations. Just broadly setting a minimum step size to the actual simulation will likely cause the simulation to require too much processing, thereby no longer matching the real-time requirement. Because of this, several different methods are introduced to provide the developer with flexibility in finding an appropriate solution for their situation.

Another related sampling issue is present in Yampa; the switching capacity of FRP causes instabilities in some simulations. Two issues are present with the switching method. The first is that the implementation inexplicably includes a delay of a single frame in the calculation. On the switch event, the behavior that is "switched into" is loaded, but not initialized. This happens first in the next frame. This can cause a slip of a time by the length of one frame. For many applications this may not cause a noticeable problem, but for others this is extremely detrimental. In particular, cases where the threshold based transition timing is used exhibit this instability.

The issues involved can be well illustrated with an example. Take for example the implementation of a simple pendulum. If we implement it, such that a change in direction (i.e. at the highest part of the arch of the arm) causes a transition to a second function that goes the other direction, the switching problem with occur. The reason to implement a pendulum in such a strange way is related to the other issue with switching. At points like the direction change of the ball, the higher order functionality is not properly simulated. Figure 8.1 shows the problem graphically. The problem is that the time delta does not match exactly to the transition point of the ball; therefore, the simulated transition of the arm will occur at a higher point in the arc than it should, causing the simulation to become instable. Experimental results show that even at 100Hz the simulation will become unstable within 5 minutes and within 10 minutes the pendulum will be looping completely! For simulations involving

higher order transitions or even discontinuities, the simulation has to sample at the moment of transition.

To make it possible to simulate such systems in a stable way, the switching functionalities can be modified and extended. As the exact moment of the transition event is what is important, the switch needs to sample at the actual moment of the switch instead of at the current time. If the switch to the new behavior occurs at the correct moment, the actual moment of event occurrence, then the simulation can be performed validly and stably. There are two basic approaches possible, a priori knowledge of the moment must be known or to subsample and find that moment. A priori knowledge is sometimes possible, particularly in causes where the user specifies the exact when of an event occurrence (e.g. at time X switch to behavior Y). The sub-sampling switching will turn out to be possible to perform transparently to the developer, due to Yampa's continuation based implementation. These functionalities are developed in Section 8.2.

8.1.2 FRVR Required Extensions

The developer using the Yampa FRP system can simulate many things with the provided framework; However, listings of required functionalities that are not present in Yampa were presented above in Section 7.6. Not only are these functionalities not supported by Yampa, but they are not implementable within the framework provided. The reason they cannot be implemented is the underlying mechanisms to Yampa's behavior (SFs) is in purposefully encapsulated in a module. Only the basic functionalities provided by Yampa are exposed to the user (see Appendix A.4.2 for details of the provided functionalities).

This means that in order to develop new time dependent SF functionalities that require use of the implementation details, the developer has to change Yampa itself. This information hiding is done so that the developer does not reintroduce the "space-time" leaks that Yampa was designed to solve. With access to the internals of the system this is a distinct possibility of the developer does not understand the inner workings of the system. For the purposes of this dissertation, adding the required functionalities to the Yampa implementation is viable, but overly restrictive. Some of the proposed functionalities are not necessarily part of the FRP concept and do not logically belong in the system. Given the open design space of the Interactive Dynamics, Dynamic Interaction and Dynamic Interaction with Dynamics areas (see Chapter 5) it seems likely that further experimentation and functional additions would be needed. For these reasons, the Yampa implementation needs to be made user extensible, so that future changes do not have to change the underlying system.

Of the remaining required functionalities to implement, only a few of them require special consideration at this level. The implementations of the time manipulating functionalities are presented in Section 8.3.3 below. Because of their experimental and unique natures, the development of the transitioning functionalities that are required are handled in their own section, Section 8.4.

In order to build all the required functionalities, a few additional supporting mechanisms have to be created. In the development, the opportunity to support the scalability requirement in an additional manner was seen and developed. Time jumps and freezing running SFs provide unique mechanisms that Yampa can provide for scaling the simulation. The principle is that sections of the simulation do not need to be calculated if the dynamic/interaction that it implements cannot possibly be active at the moment. Principally, this is envisioned as part of a scene partitioning scheme, such as BSPs commonly used in computer games. These mechanisms were developed for reducing the graphics throughput to the card, an early culling of geometry. The same principle can be used in the simulation, if support is there.

The basic question that is unclear in the simulation partitioning scheme is: what happens to the simulation when it is not running? Three basic possibilities present themselves: the behavior starts again, the behavior continues from the point it was at, or the behavior acts as if it had run the entire time. The behavior starting again method can be implemented with the switches provided by Yampa. When the behavior is switched out and back in, it starts over from time zero every time. Examples of this method can be found in many game implementations. A method that may be better in some cases is pausing the behavior when not in use and continuing it from the same state when restarted. The behavior is "unaware" of the time that passed while it was paused. The addition of functionality that allows the "freezing" of behaviors to later "unthaw" them enables such activities. Finally, in some cases, having the behavior act as if it ran the entire time might be proper. A time jump functionality allows this to be implemented. The principle here is to exploit the sub-sampling ideas explored previous. They can be used to sample the behavior in the time that has passed at an acceptable rate. This time jump behavior is also leveraged in the implementation of other functionalities.

8.1.3 Usability Requirements

Perhaps the most difficult of the requirements to achieve is improved usability of the method used. One aspect of this is the selection of an approach for the system that matches the nature of the environments that should be created. Another is matching the implementation to the developers understanding of the composition of those environments. The FRVR systems design fulfills these ideas. A more pragmatic approach can also be taken, investigating the ease of use of the system put into action.

The additional components of FRVR that were developed in previous chapter take additional steps to make the system more usable. The data types used were specified to follow the lead of a standard computer graphic mathematics library, The implementation details remain different across Haskell and C++, but are close to the same. Difficulties with the retrieval of a high enough resolution time for the kernel were also solved, with an additional interface to a cross platform module from the popular boost projects. Were improvements can still be made is in the ease of use of Yampa's Arrowized Haskell code.

Two basic approaches are identified to increase usability. A number of functionalities introduced in the FRVR implementation require the user to deal with low level aspects and functionalities of Haskell. For instance, the necessity of making IO Monad calls for data retrieve and writing means users have to understand how Monads work, a non-trivial concept even for the mathematically inclined. If these interfaces can be wrapped for the user, the system will be easier to use. The second approach is to simplify writing the program. The point-wise Arrow style (see Appendix A.3.2 in the Appendix over Haskell and FRP) makes programs easier to understand and write. The observation that the Arrows programming and Yampa programming are well explained by graphical representations leads to the idea of incorporating visual programming. Section 8.5.2 below presents the basis for a Visual Programming Environment for Arrow based code. A more in depth discussion of work performed on this can be found in Appendix B.

8.2 Improvements to Yampa Functionalities

This section highlights the changes and additions to the basic Yampa FRP abilities that were implemented in FRVR. These changes were necessary in order to make Yampa fully functional for FRVR's needs. Two issues are addressed in this section dealing with the shortcomings of existing FRP functionalities. The first issue is the sampling rate of the simulation in FRVR. The sampling rate is too slow for some simulations. The second issue is the related problem of switching behaviors at the moment of an event. These issues were discussed in the requirements section above.

Here, the implementation of those additions and changes to Yampa are presented. To address the time resolution issue three distinct components areas have been addressed: the simulation kernel, sub-sampling of behaviors, and sub-sampling transitions of switches. Each of these addresses the time resolution issue at different levels of Yampa. The sub-sampling of transitions of switches holds many synergizes with the remaining two switching functionalities that are addressed. Finally, special switches are developed in Section 8.2.4 for cases when the exact timing of the switching is known a priori.

8.2.1 Simulation Kernel Time Resolution

One of the issues of combining FRP with the VR system using the lock-step method, is that the time delta is generally too large for simulation stability. This issue was discussed in the requirements above. The solution proposed there was to sub-sample the simulation, thereby increasing the sampling frequency and reducing the size of the δt at each step. Two basic methods of sampling were proposed: increasing the sampling frequency by a given factor or creating a minimal time step. Increasing the frequency can be done by simply running the simulation X number of times per VR frame. For each step, δt_{new} is $\delta t / X$. The alternate solution is to specify a minimal time step.

The simulation then steps the minimum δt_{min} for each sample excepting for the last step, which is $\delta t_{orig} - \delta t_{min} * n$, where n is the maximal integer that satisfies the equation $\delta t_{min} * n <= \delta t_{orig}$. Here, the *deltat* provided to the simulation is denoted asδt_{orig}. In computer science terms, the final step is defined by the modulo operator: δt_{orig} mod δt_{min}.

Knowing the time deltas that should be used at each step, a solution for the actual implementation of the new simulation is required. The workings of Yampa's simulation kernel are explained in Appendix A.4.1. Since the standard Yampa simulation kernel, *reactimate*, has to be supplied with the time delta, this seems the perfect place to integrate the new sub-sampling simulation timing. The actual specification of the time delta for the step performed as part of the "sense" function. Unfortunately, the sense function may be needed by the user. This means any functionality that we build in has to be understood and used by the user. Ideally this should be transparent to the user, as a requirement was not explicitly working with the δt value. Regardless, a simple function call could be created to generate the proper values, reducing the amount they have to truly understand. Before exploring alternatives, it is informative to look at some issues that result from this sub-sampling in the context of this "simulation external" method.

There are two fundamental questions that have to be considered when implementing these sub-sampling methods: Where do the inputs for each step come from? Where do the outputs of each step go to? A Haskell/Yampa purist has a simple answer at hand, the input to the first step is retrieved, and, then, the results are all threaded to the following step. Finally, the "return" value is the return value of the final step. In some cases this may work perfectly, but this is probably the exception to the rule. There are two fundamental reasons why this is not the usual case. One is that our input and output are from the external world, i.e. the reason why the sense and actuate IO functions are part of the simulation loop. The second is that the point of this is to create and handle time dependent phenomena. Among the main difficulties here are that event occurrences only happened once, at some specific time that is likely in the middle of the interval being stepped through.

However, certain assumptions can be made based on FRVR's architecture that indicate methods of handling this that are at least sometimes valid. In particular, the method of coupling the VR system to the simulation is a key. We know that the input from the user is available for the time of the frame call. The architecture similarly asserts that only the output for the current time will be displayed. These two observations together indicated that the values of the input will not be different on each step of the simulation, at least not those from the VR side, and that only the last output will be considered. Since it is known that only one input will be available and only a single output will be displayed, the most logical thing is to provide the same input to each step and "actuate" only the last result.

Two issues remain that need to be considered. If there are events that are generated by the outside world, e.g. the VR system, those events may cause problems. As the

input is essentially replicated for each of the executions of the simulation environment, the event may be evaluate multiple times, as if it had actually occurred at each step. Generally, the design of the switching behaviors can take this into account, but it must be considered. When again considering the pure approach to using Yampa a second issue can be seen. Using this pure approach the developer may be considering the external world (e.g. FRVR's data exchange) as a global variable storage. Using the method described above, the input to each step is always the same, but the result of the previous step might actually need to be feed back in as the input. This might argue for sensing and actuating every frame. Knowing the workings of an application makes it possible to choose the right method for each situation. Since the actual needs of the developer cannot be known a priori, the ability to follow either method is left open.

As an alternative to leaving this in the developers hands, new simulation kernels can be written to replace the *reactimate* function. A version for each of the sub-sampling techniques was created: *reactsubimate* and *reactminimate*. Figure 8.2 shows how these two kernels execute. Both receive a single input each frame, just in the case of *reactimate*. The input value is given to each sub-step, and the appropriate portion of the δt given is provided each sub-step. Only the return of the final step is returned. For the developer using these functions there is only a single difference in their setup, each requiring an additional parameter. *reactminimate* requires the minimum step size (a *DTime* value), and *reactsubimate* requires the sampling factor (an ***Int*** value).

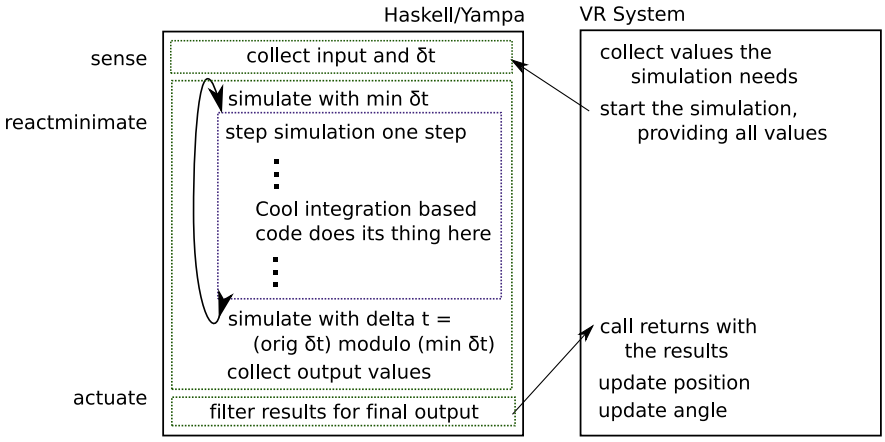
8.2.2 Sub-sampling Behaviors

In large simulations, it is probable that portions of the code need to be simulated at a higher sampling frequency than other parts. One way of assuring the necessary sampling frequency is through use of the either *reactiminimate* or *reactsubimate* kernel presented in the prior section. This has a distinct disadvantage, as the entire simulation is sampled at the higher rate. Portions of the simulation may be oversampled using this method. Beyond the slowdown that the oversampling causes, errors may be introduced either because of oversampling or because each run of the system will have the same input. The better alternative would be to introduce a similar sub-sampling functionality at the level of individual behaviors.

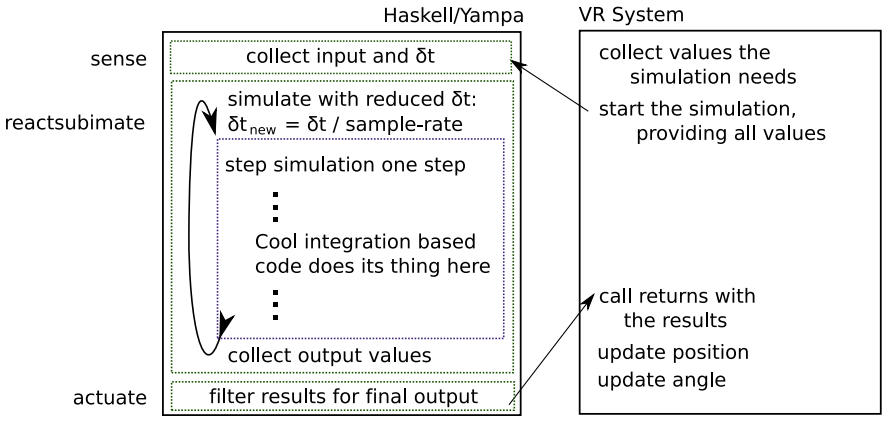
The same two sub-sampling methods that were investigated at the kernel level are applicable at the level of behaviors. The only difference to the modified kernels is that these functionalities provide a higher sampled environment inside of the complete simulation. The behavior *subsampleSF* provides sampling of a sub-tree of the simulation at a higher rate, for instance $10x$. The *subsampleSF* is simple for the developer to use:

```
subsampleSF :: Int -- subsampling number, how many samples per delta
            -> SF a b -- SF to sub sample
            -> SF a [b]
```

The sampling rate and a *SF* to be run in the created environment are inputs to the function that wraps the *SF* in a new sub-sampled environment. The same input is used



(a) reactminimate



(b) reactsubimate

Figure 8.2: The sub-sampling simulation kernels: a) shows the kernel based on maintaining a minimum time step. b) shows the kernel based on sub-sampling the δt.

in every run. However, the sub-sampling environment wrapper differs from the kernels in that all of the output is returned in the form of a list.

The major difficulty with the design of sub-sampling behaviors is precisely what to do with the output values of each step. In contrast to the total simulation, it is not clear how to correctly handle the values. The values may be passed on to somewhere else, not just placed in the data exchange module. Because of this, the sub-sampling behaviors created return the complete list. A simple behavior can be used just select the last value if desired. The actual usability of these values is highly limited, as the actual δt used in each step is not available to the programmer.

Sub-sampling the behavior using the minimum step size approach is supported. The *subminsampleSF* function works similarly to the *subsampleSF* function, just taking a minimum step size instead of a sampling factor. Like *subsampleSF*,

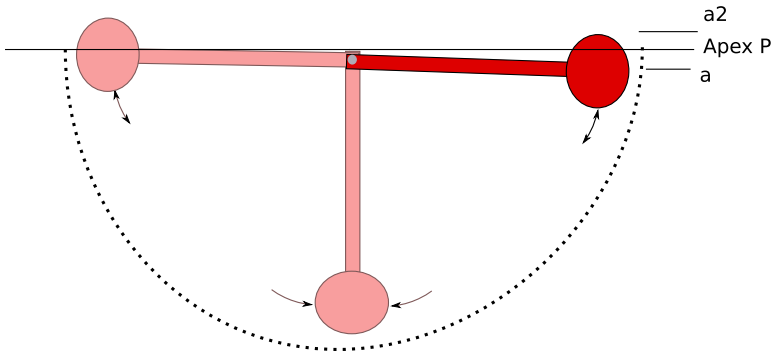


Figure 8.3: This diagram demonstrates the "time leak" produced by sampling higher order integral systems at too large a resolution. The example of a pendulum is used to highlight this and further to pendulums can be found in Section 9.2.2

subminsampleSF returns a list of results. However, it is important to realize that the number of outputs in the list is not deterministic. It can be guaranteed that at least one output will be present, but in the case of a small minimum and a large δt the number of outputs could be very large. Given that Haskell is functional language that provides many list processing functionalities, handling this is not necessarily a problem, but the developer must explicitly handle the case.

8.2.3 Sub-sampling Switch Transitions

The sub-sampling methods presented in the previous sections address an issue with the sampling frequency of behaviors, in particular that they are not sampled at a high enough rate. Related to this is an issue with the timing of switching between behaviors in the reactive portion of Yampa. Events in Yampa are assumed to have occurred either at the instant of the last simulation step (in the case of the external switch, *rSwitch*) or at the time after the application of the current simulation step (the internally generated *switch* and *kSwitch*). Externally generated events in the VR system can easily be handled as per the semantics the developer wishes via these mechanisms. However, simulation triggered internal events are also assumed to have happened at the instance of one of the simulation steps. The timing of the event sampling is of critical importance for some simulations, particularly physical simulations that are dependent on higher-order transitions.

A simple example to highlight this issue can be framed by simulating a pendulum. The physical equations can be used to create the behavior (the example in Section 9.2.2 describes the simulation in detail for those unfamiliar). At the apexes of the arc of the pendulum, the angular velocity of the pendulum crosses from positive to negative or negative to positive. The diagram of Figure 8.3 illustrates one side of the path. The formula takes this into account, but the physics based formula assumes a symbolic integration. In terms of numerical integration, this means the step size approaches zero,

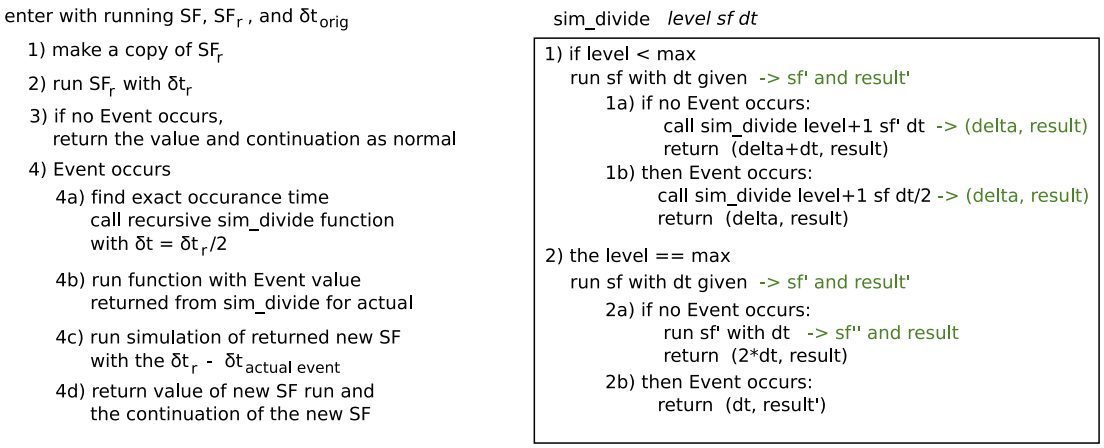


Figure 8.4: This figure illustrates the algorithm used for sub-sampling the switching event. On the event occurrence, the time delta is divided and sampled again until a specified level, performed by the recursive sub-divide function. Finally, the new SF, generated from the result of the more precise Event occurrence, is run with the remaining time of the original δt.

such that an integration step happens at the exact moment of the velocity change. In contrast, the Yampa integral is dependent on a δt that tied to the simulation loop and much larger than zero. The step size is only an issue at the transition points. If we assume the sampling occurs at point a in the diagram the velocity remains positive. In the next frame the pendulum travels to point $a2$ on the next step, overshooting the true apex P. A zero crossing of the velocity occurs, but is not applied until the next frame (integration is delayed, but if it wasn't the negative velocity would be applied to a causing a similar problem in the other direction). In the next frame the simulation moves downward, but from point $a2$, not the apex point, P. Collectively, this cause the simulation to "speed" up. This is essentially the "time leaks" that Yampa attempted to fix.

The sub-sampling behaviors of the last section partially address this by making the time step smaller and thereby reducing the overshoot. However, this doesn't really solve the problem and causes excessive calculations for all other points. Another possible solution is replacing the integration function with an adaptive algorithm, for instance a Runge-Kutta integration. This solution is more acceptable. It handles the case of physical simulation well, but for any simulation that is not based on higher-order integration it will also fail. Instead, a method that samples the behavior at the moment of the transition is desired. The subsampling switches are designed to alleviate this problem by re-sampling the time of the event multiple times to converge on a more accurate event time.

Conceptually, the sub-sampling switches are implemented by exploiting the continuation based SF design. At each step, the running SF is executed as normal. The return value is inspected for an Event occurrence. When an event occurs, we know

that the event happened at some time during the interval of the δt. Because of the continuation based implementation of behaviors, it is possible to hold onto the SF in its state before running it. Exploiting this continuation nature, we can resample the SF with a smaller time step. Although different versions of the sub-sampling switches exist, the most efficient is the *switchTSampled*. This version works by a recursive divide and conquer method. The time sample is divided in two and the continuation is rerun. If the event is thrown, it is rerun again with the time again halved. If the event is not thrown, the half of the current δt is added to the current δt. In this manner, the algorithm converges on time of the actual event occurrence. The algorithm followed is presented in pseudo code in Figure 8.4. This recursion depth is specified by the developer. Even small depths of four lead to better resolution of the exact timing than using the behavior sub-sampling of the last section and even of the adaptive integration methods.

The second aspect of these sub-sampling switches differs from the original switch functionality(see Section A.4.1 for details of the switches). As noted before, an issue with the standard switching method is that the new SF is not actually started until the next frame. In order to effectively sample the exact moment of transition, the transition needs to actually occur at that moment and the new SF should be run from that point. The sub-sampling switches initialize *and* run the new SF with the remaining time of the original δt.

8.2.4 Time-aware Switching

For a special class of behaviors, an alternative method of getting exact switching can be used. These behaviors are explicitly aware of the time of the switch at all times. The simplest of these encapsulate the idea of a switch at time $10.0s$; this is equivalent to after $10.0s$, since each behavior operates in a local environment that starts at time $0s$. Using the standard Yampa mechanisms, the *after* SF, the switch occurs at the first time greater than 10.0s. In such cases switching at the proper time is easy since it is explicitly known when it should happen. Less obviously, behaviors such as the keyframe animation behavior that will be presented in Section 8.3.2 are also of this class of predefined length behaviors. Since the systems are aware of when the actual switch event occurs, they can be extended to inform the switching mechanism of its timing. A specially built switch can then switch into the new SF at the actual moment of switching, like the sampled switches above, but without any extra computing overhead and no side-effects.

This functionality is implemented in the wrapper behavior *switchTimed*. *switchTimed* derives from the standard *switch*, the same can be done with any of the switch types. *switchTimed* is defined as:

```
switchTimed :: SF a (b, Event (c, DTime)) -- initial behavior
    -> (c -> SF a b) -- generating function, creates new behavior
    -> SF a b
```

The only difference the developer sees when compared to a standard *switch* is that the Event return type has to be extended to return both the standard transitioning value and, additionally, the remainder of the time delta. This is a simple case for something like keyframing; the total time of the animation is known a priori and the remaining δt of the last frame can be easily calculated. On a switching event, the generating function is called with the appropriate part of the Event data, just as in the standard switch case. The new SF is then initialized with the same input and run with the remainder of the time delta provided in the Event data.

8.3 Extended Basis Functionalities

A number of DIVE functionalities are not present in the Yampa implementation. In this section, functionalities are developed to address these shortcomings indentified in Section 7.6.3. The functionalities are divided into four subsections.

The first subsection handles a major issue of Yampa in building the remaining extensions: Yampa is not designed to allow user extensions. The extensibility of the system was identified as a system requirement in Section 7.1.2; however, Yampa is designed to deny this access. For extending Yampa to meet FRVR's requirements and for future development, this had to be addressed. The next extension demonstrates how tradition computer animation tools can be integrated by providing support for the popular keyframe animation technique. The third sub-section deals with time manipulation. The major component of this is time skewing, i.e. allowing the simulation to run at a speed other than the physically real "clock time." Finally an extension which enables the critical undo functionality that permits good usability is developed and presented.

8.3.1 Extensibility of the Yampa Modules

A core issue to creating extensions to Yampa's functionality is that it was explicitly designed to inhibit the user's ability in this regard. The reason being: if the developer can get at the internals of the Yampa system, they could very easily reintroduce the "space-time" leaks that the Yampa implementation was designed to solve. The creation of the extensions necessary for this dissertation, required that a modified version of the Yampa code base had to be created (Yampa is distributed under a permissive license that allows modification and re-release). Such a solution is viable for FRVR to function, but does not truly address the extensibility requirement.

To ensure usability and extensibility, a new version of Yampa had to be created. The modifications to Yampa were a combination of user extensibility and the different functionalities discussed in this chapter. A re-evaluation of the structure of the Yampa "AFRP" module led to a structuring of the project. These structural changes better componentized the code base, opening up the internals for extension, and improved the future maintenence of the code.

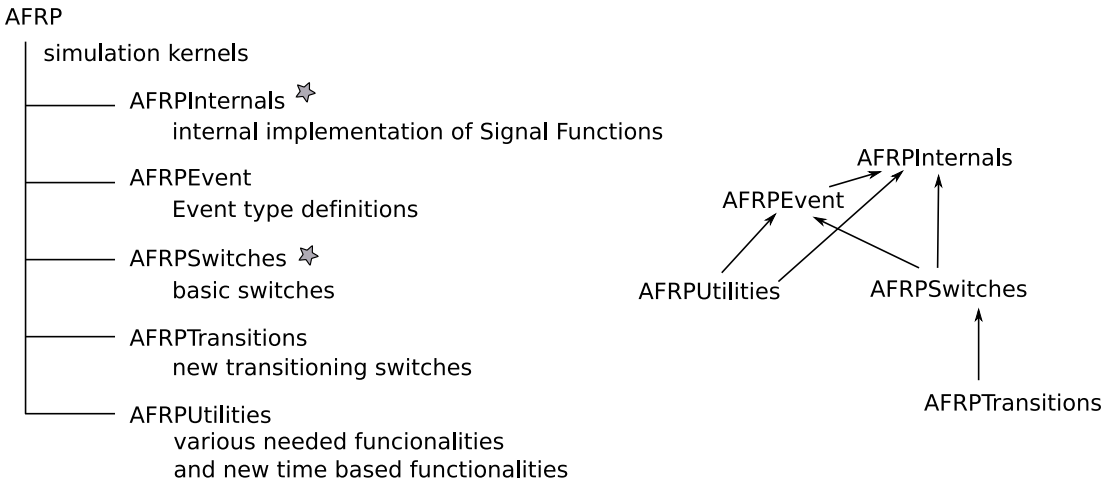


Figure 8.5: The structure of the refactored Yampa modules is shown, with their relationship demonstrated. The left side lists the functionalities that each modules contributes. Those marked with a star are new files, created primarily out of existing code. The AFRPTransitions module is completely new content. The left side shows module dependency.

Three basic changes had to be made to the code base structurally. While some structuring of components was present, many aspects of the system were found in one file, `AFRP.hs`. The primary reason for this is that all of the behavior (SF) related internals were defined in that module, but not exported. As a result even the other Yampa sub-modules were unable to access these internals. Interestingly, the Event type was separated out and a number of additional functionalities that did not require the SF definition were in individual sub-modules. The code structure was improved by:

- separating out the system internals into an *AFRPInternals* module,
- separating out the switch functionalities into their own module, *AFRPSwitches*, and
- sorting the remaining code into the proper places.

The final structural change was the addition of a module for the transition functionalities to be developed in the next section. The resulting structure for the base modules is shown in Figure 8.5 as well as their inter-dependencies.

The restructuring places the definitions of the basic types (*SF*, *sfTF*, *integral*, *DTime*, etc.) into a single module interface, *AFRPInternals*. This module is imported into the AFRP interface, but only the portions that are required are exported, just as in the original AFRP interface. The advantage, beyond a better structure in the Yampa code itself, is that this module is available to advanced users. Standard developers should only need the AFRP interface, as they should not extend the time based SFs. However, advanced developers can explicitly import the internals module when extension of the functionality of Yampa is required.

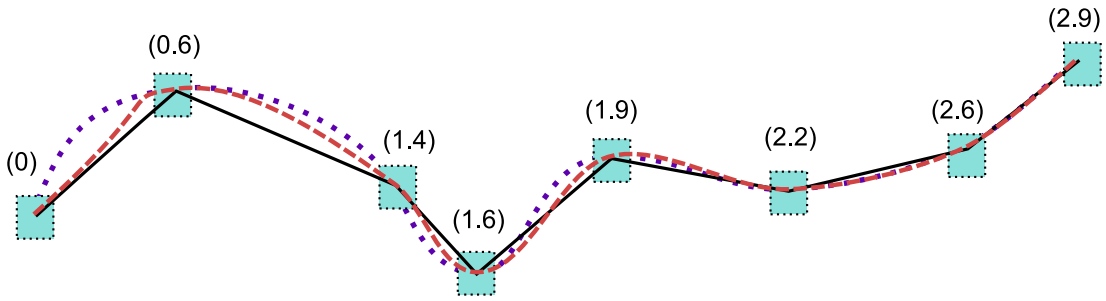


Figure 8.6: The illustration shows keyframed 2D positions with the times for each keyed position listed. Three different possible interpolations for the values in-between the keys are shown.

8.3.2 Traditional Animation Tools Support

The inclusion of dynamics is common to many areas that are based on computer graphics. Standard techniques for the creation of dynamics come under the term animation. Animation has been discussed in Section 3.3. Unfortunately, these techniques largely rule out the possibility of interaction, making them ill suited to the focus of this work. However, the enabling of what is a standard method for the inclusion is important in the contexts of this work. Because of its general utility for dynamics creation, the integration of keyframed animation techniques into FRVR has been undertaken.

Keyframe animation is a pseudo standard method in the CG industry, largely because it is easy to understand and very versatile. Even animation tools like inverse kinematics typically rely on such techniques for defining where the avatar should move to. In VR their keyframing is often used for less obvious dynamics, like the playback of recorded sessions, particularly in the creation of "canned demos." For such demos, an expert user navigates an environment and their path is recorded at points along the way. Automated demos then play back the path they took based on the keyed points that were recorded. Other possible dynamics were discussed in the Dynamics Chapter (Chapter 4).

Keyframing is composed of a collection of data points and algorithms to create the dynamic from the data. Figure 8.6 illustrates the principle of keyframing. The "keys" in animation are the data points associated with a given time. These paired values (for instance 3D positions) and times indicate that the animation should reach that position at that time. Strung together they define a path over time. The algorithms are required to create the movement between the points, such that at the provided times the animation reaches the right values. There are many algorithms, some of which will be directly or indirectly supported here. For further information to these animation methods, please consult one of various books on the topic [JNW06, Par02, PR02, Vin03].

Adding basic support for keyframe animation to Yampa is easy. Lists of paired keys and values are a specialty of Haskell, time dependent behaviors is Yampa's specialty. All that is needed is the proper framework to be built for the most important compo-

nent, the interpretation kernel. Here, the implementation can benefit from Haskell's polymorphic handling and higher-order function handling. Instead of writing special keyframers for different values, the interpretation kernel can be given over to the behavior. Naturally, time is handled transparently to the user. The basic animation SF is defined as:

```
keyedAnimation :: RealFloat b => (a -> a -> b -> a)
        -- interpolation kernel: in1 in2 trans-time
        -> a -- initial value (i.e. key at time 0)
        -> [(DTime, a)] -- time delta and Keys
        -> SF c (a, Event ())
```

Using this type of function the basic interpolation functions can be easily programmed. The output has been expanded to be not only the keyed output, but to include an Event. The event is thrown when the animation plays the final key. The structure of this matches perfectly the *switch*, allowing easy integration of behavior changes after completion of a keyframed animation. After reaching the end of the keys, *keyedAnimation* simply returns the final key indefinitely.

A common extension to keyframed animations is to create small animations that repeat and simple "loop" them. For instance, this is a standard method for the creation of avatar motion, the "walk cycle." Rather than requiring a new behavior to achieve this, Haskell's lists and lazy evaluation can be take advantage of to create animation loops. This can be achieved by simply looping the input, i.e. creating an infinite loop. This infinite loop can safely be given to the *keyedAnimation* behavior and will run forever. The lazy evaluation ensures that this works. It is also advantageous from a conceptual point, that this method makes explicit the workings of cyclic animation.

8.3.3 Time Manipulation Functionalities

In the discussion of Dynamics in Chapter 4, the idea of time being malleable in the context of a computer controlled environment was introduced. The synthetic nature of the environment allows the flow of time itself to be dynamic. As the user's frame of reference is the real world time, such a functionality is generally referred to as time skewing. Time skewing of the simulation can be performed in FRVR in three manners. The simplest case is to skew the entire simulation. This can be done at the highest level of the system, by directly manipulating the time delta given to the simulation kernel, e.g. *reactimate*. The second and third functionalities are SFs provided through extensions presented here.

These two functions operate by skewing the time delta for each step by some given value. The two functionalities differ in how and when the skewing factor is specified. *timeSkewSF* is a function that takes a floating point value for the skew factor. The function wraps a given SF, such that the skewing factor affects it (and its subparts). This is a "statically" defined skewing factor defined as the SF wrapper is generated. The second version of the behavior, *timeSkewSFd*, uses a dynamic skew factor. Instead of the wrapping SF being generated with a specific skewing factor, the SF's input is

extended to take an additional parameter that controls the skewing factor each frame. The behaviors are defined as:

```
timeSkewSF :: SF a b -> Double -> SF a b

timeSkewSFd :: SF a b -> SF (a, Event Double) b
```

Listing 8.1: Time-Skewing Function Signatures

In the definition of *timeSkewSFd*, we see that the second parameter, the skewing factor, is given inside an *Event*. This follows a precedence style already set in Yampa. This mechanism allows the value to only be specified when the value changes. An event occurrence is tagged with the new value. When the value is *NoEvent*, the skewing factor remains the same as the previous frame.

The implementations of *timeSkewSF* and *timeSkewSFd* are straight forward. Since they are defined with respect to the wall clock, the time delta must only be scaled by the factor. This method is the typical method used in other systems, though in this case it is completely hidden from the developer. Beyond the ease in not having to deal directly with this, the handling of special cases is done internally. However, those factors should be addressed here, as they are important details to the implementation and also some of those details need to be understood by the developer.

The time skew factor is specified in both *timeSkewSF* and *timeSkewSFd* as a double value, since the time delta is a ***Double***. The scaling of the time can be any value, except time may not flow backwards, i.e. the value must be non-negative. A zero scaling may also be used, effectively stopping the running of time dependent behaviors.

Several aspects of using the time skew behaviors are importance to note. Time skewing, if used with consideration, can have very adverse effects on the underlying simulation. A zero scaling stops time, but calculations are still performed with $\delta t = 0$. Yampa is designed to handle 0 time deltas, and Yampa assures that a divide by zero does not happen. However, the code still is executed. Values dependent only on time do not change. What does change, however, are values that are dependent on interactions, i.e. input values and Events are still processed.

Another aspect of time skewing that the developer must consider is that it does not affect the sampling rate. Time skewing works by directly affecting the resolution of each step. For instance, if time is slowed down by a factor of ten for the popular "bullet-time" effect, the time resolution will also be a factor of ten smaller. This leads to oversampling. Conversely, speeding up the simulation, for instance to skip the "boring parts," means that the time delta is enlarged. Correcting the undersampling can be performed by using the sub-sampling behaviors found in the previous section. As it cannot be known a priori what method is best for a particular simulation, the programmer is left to assure the correctness of the simulation when using time-skewing. For dynamic skewing cases, the usage of *subminsampleSF* is suggested for a robust simulation.

8.3.4 Undo

An undo functionality is seen as one of the crucial functionalities required for good usability of systems. For that reason it is interesting to find that none of the systems reviewed in the research for this dissertation supported that functionality. The FRP paradigm also ignores the undo functionality, even though the Yampa system was designed for GUI creation! The exact reason that this is missing is not known, but a portion of that might be the difficulty undo presents in the context of DIVEs.

The implementation of undo in a desktop environment is fairly easily created due to the common development paradigm, event based systems. Since all interactions are viewed as events, undo is simply a recording of the events or, if reversing the action is not possible, the state prior to the event occurrence. Given that many of the systems reviewed (see Section 6.2) work on a principle of all interaction being events, this method should be possible to perform. The main issue that makes DIVEs in particular so difficult comes out of the area of Dynamic Interaction presented in Section 5.1. Dynamic Interactions are continuous in nature. This means that every frame those interactions have an effect and must be handled. This makes the creation of an undo using the event method intractable, as every frame must be considered an event. A natural way to attempt to bypass this issue, is to simply save only at certain points. However, this is not possible in the case of many dynamic interactions, as the reaction of the system to the input is dependent on every moment of the continuous interaction. The method used as the capture the complete interaction and dynamic in its context to function properly.

Yampa's design enables a functionality to be created that can capture the complete behavior of the system and at any point in the simulation. The continuation based stream implementation of Yampa makes the implementation of the undo functionality straightforward. FRVR introduces two undo functions into the system, *undoSF* and *undoStackSF*. Both of these are based on the premise that undo is a stack operation. The difference between the two undo behaviors is that the *undoSF* limits the stack size and *undoStackSF* allows an unlimited stack size. The undo behaviors create special wrapper environments around other SFs and are easy for the developer to use. They are defined as:

```
undoSF :: Int -> SF a b -> SF (a, Event StackEvent) b

undoStackSF :: SF a b -> SF (a, Event StackEvent) b
```

Listing 8.2: Undo Function Definitions

The undo behaviors react to events that are tagged with the *StackEvent* type. *StackEvent*s are either a *Push* or a *Pop* ***Int***. The reason for the integer parameter of the pop of the stack is to allow the user to go back multiple saved moments in a single jump.

The actual implementation works as would be expected. On an *Push* event, the continuation is frozen and saved to the stack (a description of this process can be found

in the Appendix A.4.1). A pop of the undo stack retrieves the proper frozen SF from the stack and returns it as the continuation for the next step after "unthawing" it.

8.4 Transition Handling

The design space of Interactive Dynamics was identified in the analysis of the design space of DIVEs as one of the areas of high potential, but one where little exploration of the design space is known. In the investigation of Interactive Dynamics in Chapter 5, the standard method for such interactions was presented: the turning on or off of dynamics. This is, however, a rather uninteresting method. It can be conjectured that the reason few true Interactive Dynamics exist is due to the limited support for their implementation. However, even conceptual it is uncertain what should be implemented. The area is simply not well explored yet.

However, one potential method of better supporting the interactive dynamics can be identified. If one views the Interactive Dynamics as a series of behaviors that change based on the interaction, then this area fits well[1]. The focus of the area, in a simplified view, attempts to gracefully handle the change between the previous behavior and the new behavior. The transition handling approaches of computer animation are this sole area to have identified methods for handling the changes between behaviors.

This section addresses the work performed to adapt and extend the ideas of that area to the more general case of interest for DIVEs. The section will begin with a look at the different aspects of handling these transitions. After gaining a conceptual understanding of the area, the developed extensions to the Yampa system are presented. The extensions are grouped by the different transition timings and methods they support. It should be noted that the work of this section, should not be considered the final answer to the issues of programming such Interactive Dynamics, but instead these methods provide a structured support framework for the exploration of this interesting area .

8.4.1 Conceptual Foundations

This section focuses on the conceptual foundations of handling the change from one behavior to another in the face of user interaction determined timing. In the first subsection, the issues of transitioning at a non-deterministic time are highlighted. This discussion bases on the understanding from the sole community to have addressed these issues, the computer animation community, and expands on their approach to include a more general view. The remainder of the section, looks more in depth in how these issues can be handled appropriately drawing on the knowledge of the computer animation area. The two main approaches that can be taken are each explored in its

[1]This type of Interactive Dynamic concerns itself with discrete interactions. The other set of Interactive Dynamics fall into the catagory of Dynamic Interaction with Dynamics and is more difficult yet to conceptualize and implement. Chapter 5 covers the design space and implications of these topics.

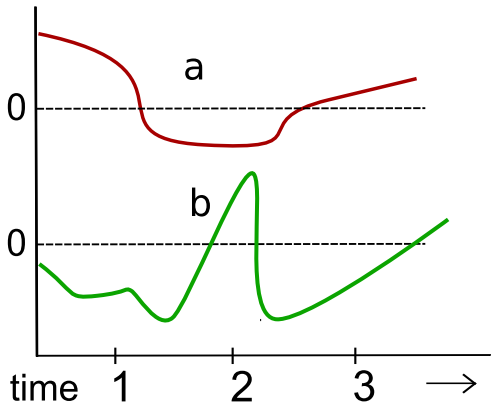


Figure 8.7: Two Example Behaviors, a and b.

own right in the following sub-sections. The discussions of this section are limited to the issues at a conceptual level. The implementation issues are handled in the following sections.

Understanding the Issue

The discrete interaction with dynamics is currently the most likely way to support the development of Dynamic Interactions. However, even performing this is not necessarily easy and definitely not well supported. The FRP paradigm was selected as the basis system for this work in part because it matches so well the abilities needed for precisely such activities as Dynamic Interaction. In FRP, the switching between behaviors is the major method to create Interactive Dynamics and also to handle changing behavioral structures over time in the dynamics. Where support, and even conceptual understanding, is lacking is in the handling the transition from one behavior to another at the point of interaction. This section looks at the issues that make this a difficult problem and also introduce the ways support can be introduced.

In general, the transitions between behaviors is an undeveloped area. A survey of the same research directions as surveyed in Section 3, reveals that this is largely ignored across all fields[1]. The single community that has made some study of this problem is the animation community. The following look at transitions will largely derive from the animation community's viewpoint, though we feel these methods are broadly applicable.

[1]It is important to note that some of those communities use the term transition to mean: the function which defines a change from one state to the next. Most notable in this regard is the simulation community.

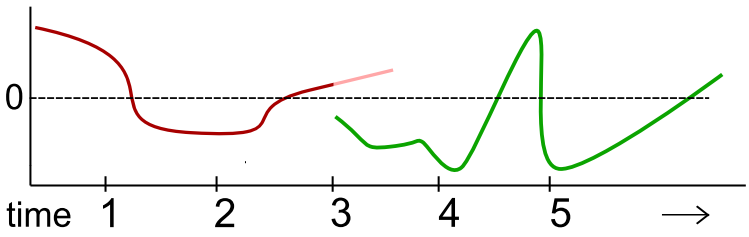


Figure 8.8: AFRP Switch Transition from a to b at Time $t = 3$.

The issues at hand are best explained on hand an example. The discussion throughout the section is loosely based on a familiar case to many, an autonomous character whose movement changes as the user interacts. Assume that we have a behavior - a continuous function over time that defines some dynamic - that is running. At some point, an event signals that the behavior should change. Figure 8.7 shows two functions in time, a and b, that represent the behaviors. The major question is, what should the behavior be, if behavior a is running and the event occurs at time $t = 3$? The character should follow behavior b after the event, but how this is handled is in question.

The easiest answer, and the one that is directly supported by FRP and most other systems, results in an output that is illustrated in Figure 8.8. There, one sees illustrated how this immediate switching method can easily lead to discontinuities in the dynamics that are most undesirable. In some cases, this may not to be an issue, such as when the event is internally generated at an appropriate moment. However, recall that the issue being dealt with is interaction, meaning that the timing is initially controlled by the user. When such a case occurs that an immediate switch does not cause a discontinuity, it is generally due to a clever programmer/designer. However, even cleverness can not be assumed to fix every case.

In the literature, there are two over-arching methods to handling the transitions, such that discontinuities do not occur. These methods both come from the animation community and specifically deal with the handling of animations of characters. The complete set of motions that the character can perform are available in the form of a set of animations of the characters geometry. These animations are generally short animation cycles, e.g. the walk cycle. The animations are quite like behaviors seen in Figure 8.7. Players of early computer games will also recall seeing their character jump between positions, when they indicated a change in behavior, just as illustrated in Figure 8.8.

From that community, two approaches can be found. One method is to assure higher-order continuity, e.g. instead of just dealing with the position, deal with the higher-order value, the velocity. These methods require the author to consider the problem and provide a fix for it. Generally it involves slight modification to both behaviors, but may not have to be specially rewritten. The second method is to constrain the transition such that either the timing of the transition occurs at points that create

acceptable changes or the values of the transition are transformed through special handling the transitions appearance in a way that is acceptable. These methods do not generally require modification of the underlying behaviors, but instead rely on other methods to achieve an appealing appearance. These methods are presented individually in more detail in the following parts.

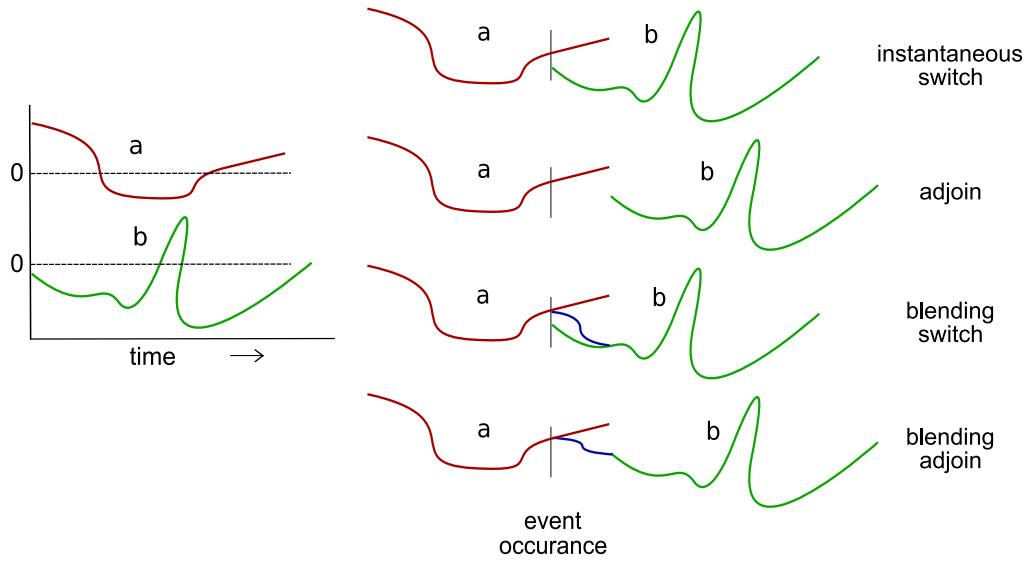
Higher-order Continuity

One of the ways of ensuring continuity of the values being generated across the transition is by ensuring higher-order continuity. In the realm of time dependent continuous functions, this means dealing with the values that are integrated instead of just the generated values, e.g. the velocity or acceleration. When the system is based on a physical model, this is a very natural approach. However, even with non-physics parameters this is possible. This usually involves determining the higher-order constraint of the behavior that is being transitioned from, based on its history, i.e. the derivative of the first function as it transitions.

A number of examples of this method are demonstrated in the examples of the next chapter. The physically based simulations have obvious implementations. Interestingly the autonomous entities category of Dynamics, particularly the Simple Beings subcategory, is commonly implemented in a manner that is compatible with this method. The classical implementation method, Boids, essentially does this by operating per frame only on the "desired acceleration" (a description of the Boids method can be found in Section 9.4). By operating only at this second-order level, the method guarantees continuity at the level of position and orientation and even velocity of the entity.

In implementation terms, higher-order continuity methods generally require programming by the author. This is largely because the higher-order continuity has to be developed from the beginning and cannot be extracted from the behavior internals. It is possible, based on the output values, to recalculate the velocity or even the acceleration of an entity through the derivative. This is very computationally costly and is not particularly accurate. As Yampa provides a derivative method, this approach can be taken. The behavior implementations in FRVR are generally defined via higher-order functions, i.e. integral calculus. To enable continuity across the transitions is in many cases possible, since the switch mechanism allows the exiting function to simply export its working values, e.g. the acceleration and velocity of a moving object. The developer can then either use this value to initialize the next behavior or to build a constrained transition in a similar manner to that discussed in the next section, but at the level of the higher-order value.

An example of this can be found in the transitioning pendulum of Section 9.2.2 and the Newton's Cradle example in Section 9.2.4. By passing the velocity of the ball when striking the remaining balls to the new ball, the continuity of the movement is assured. The Boids example in Section 9.4 demonstrates a second approach to this, where the behaviors that change operate only on the desired acceleration.



(a) Reference signals (b) Possible transitions of the signals on event occurrence

Figure 8.9: Possible Transition Timings

Constrained Transitions

To control the transition between the two behaviors, a method that is external to the behaviors can be used. This has the advantage of not having to modify the behavior itself to make transitions work. The disadvantage is that the methods typically rely on making "it look good," not necessarily making a meaningful transition. To understand how to do this, first an investigation of the ways in which the two behaviors could be combined together, assuming an event at a random time causing the switch, has to be undertaken. Based on the possible ways in which the behaviors can combine the methods of constraint can be highlighted.

To better understand the combination, the same example as above forms the context of the introduction to constrained transitions. To begin with we will assume the classical character animation style, where the animation is an interval defined behavior that is looped. Using this behavior style is significant, as it means that there is a definite end to the behavior, before starting again. After presenting the different concepts that are followed in computer animation, a discussion of the extension of these to the general case is given.

Figure 8.9(b) displays the possible transitions, based on the signals shown in Figure 8.9(a). This is the basic set of transitions that can be made out of switching from a to b. These basic transitions types we term: instantaneous switch, adjoin, switched blend, and adjoin blend.

Putting these transition methods in the context of classic character animation provides a better understanding of them. We will assume that the simulation is animating

a "walk cycle" of a character to start with. An event is triggered and the character should change to a "sit" behavior. This interruption comes in the middle of the cycle, and the walk animation starts with the legs in the middle. In the next paragraphs, each of the four possible transitions is explained via this example.

The *instantaneous switch* is an obvious transition method, and a method already present in Yampa. In our example, this causes a discontinuity, as the character springs - in mid stride - to start its sit animation. This method was commonly used in the early days of real-time 3D computer games. Because of the visually displeasing nature of this, other methods are now used.

Adjoin extends the switching to be context sensitive. Adjoin introduces the ability for the executing animation to continue until it reaches a point where it is valid to switch to another animation. In our example the walking character would end the walk cycle, ending in a standing position, and then start, from that standing position, to sit. This method essential moves the task of creating continuity off-line, to the animator. This remains a common approach in the animation of characters. Generally, all character animations are created as cycles, where the cycle starts and ends in a specific position, e.g. for our character example this is typically a neutral standing pose. By doing this, the animator/designer can enforce continuity of the animation, though at the cost of having the reaction of interactions delayed. The other cost is that the animator must get all animations so that they can be switched from one to the next; a task that is not easy. Most gamers are familiar with this type of animation as it is still commonly used.

The final two methods, attempt to enforce continuity in the transition through the addition of external smoothing. In moth cases this is achieved by adding a period of transition, where a smooth transition between the values produced by both behaviors is achieved. The difference in the methods is in the timing of the start of the second behavior's execution. The switch version is an instantaneous start and the adjoining version starts first at the end of the first behavior. In the computer animation community this approach is common and often denoted as "ease in/ease out." This method can be found in any computer animation book, for instance in Chapter 3.2 of [Par02]. The basic idea is to blend or morph between the output of the two functions. Hence, this method is also denoted as blending or morphing. In our denotation a *blending switch*, allows one to implement the classic "ease in/ease out" functionality.

Key to these blending methods is the interpolation kernel. These kernels do the actual blending. They interpolate between the two values given by the behaviors. Interpolation kernels were discussed in regards to keyframing in the prior section, but the interpolation here has a slightly different purpose. Here, the purpose of the interpolation kernel is to: smoothly move from only the value of the first behavior at the beginning of the interval it is defined for until the end of the interval where the resultant value is only the value of the second behavior. Typically, the interpolation function does this smoothing by weighting the percentage of the resultant value that comes from each of the behaviors, i.e. at the beginning the first behavior contributes

100% and at the end the second behavior contributes 100%. The difference in the kernels is how the percentages of the values contributions in-between are calculated. The typical interpolation kernel for these blending functions uses some sort of asymptotic function, such that the continuity appears to be higher-order (or at least not jerky).

It should be noted that the *adjoining blend* is just a special variation of the blending switch. Here the second function adjoins with the end of the first, as in the adjoin case; however, the first output value of the second function is used for the blending interpolation kernel. This method overcomes a limitation of the original adjoin function. In the original adjoin, assuring the continuity of the functions at the adjoining moment was left to the user. The adjoining blend is only practical in the special case illustrated by the cyclic behavior where the first behavior has a specific and known ending point. Further discussion of these points is given in Section 8.4.4.

Up to this point, we have described four different possible transitions based on the approach as it is used in classical character animation. The adjoin cases shown are dependent on the fact that the behavior can be seen as a distinct, known interval and that the animator has taken care of the continuity at that special transition point. Unfortunately, in the general case, the behavior is likely a continuous function, defined over the time ($[-\infty$). At best, that behavior can be expected to sporadically have transition points. The result of this is that the timing of the valid transition ends is likely not known a priori. Since the more constrained interval versions are simply special cases of the general case, the focus of the implementations below is on the general case where a priori information is missing when possible.

8.4.2 Adjoin Transition

The implementation of the adjoin transition can be performed in a number of ways. In some cases, through a bit of clever programming, the programmer could manage to build an adjoin transition with the tools already present in Yampa. However, this is neither a straightforward nor a particularly efficient implementation, so the inclusion of a method that handles this natively is advantageous. Of importance in the implementation is to support the case where the timing of the "safe" to transition points in the initial behavior are not known a priori.

The adjoin functionality can be addressed most easily within the Yampa FRP design in terms of a "double switch." This approach is based on the observation that the actual switching point is defined by two events in combination. Both the condition of the behavior transition interaction event (the external event from here forth) and the condition that the original behavior is ready to switch are important. The internal event is only important after the external event has occurred. The basic algorithm is: on external event, keep running the original behavior until it throws its switching event.

This functionality implemented in the *switchWait1st* SF. The *switchWait1st* implementation is a combination of the internal event generating *switch* and the external

event generating *kswitch*. The initial SF and the SF that detects the external events are the same as in the *switch* and *kSwitch* respectively. The only deviation from the strict combination of the two, is that the SF generating function is modified. *switchWait1st* is defined as:

```
switchWait1st :: SF a (b, Event d) --starting SF
   -> SF (a,b) (Event c) -- SF that detects the occurrence of the event
   -> (SF a (b, Event d) -> c -> Maybe (SF a b))
      -- generation function
      -- if a new SF is given the following happens:
      -- the SF will be run AFTER the 1st SF triggers an event
   -> SF a b
```

Listing 8.3: Adjoining Transition Behavior: switchWait1st

The new generating function follows loosely that of the *kswitch* switch. That function received the continuation of the initial SF and the value of the tagged external Event trigger. The function was to decide, either to generate a new SF based on the provided value, or to return the continuation, i.e. continue the initial SF. The major issue with keeping this functionality is that the typing of the initial SF between *switch* and *kSwitch* do not match. Instead the function is modified to return a ***Maybe** SF*. In order to keep the initial SF running and the switch also, the function simply returns ***Nothing***. If the switch should react to the external event, i.e. switch, it returns the new SF to be switch into in the Maybe. When a SF is returned, it will be started, *after* the next time the initial SF throws its event. At this point *switchWait1st* functions the same as (and is implemented as) the original *switch*.

A second version of the adjoin transitioning behavior is the *switchWait1stTimed* behavior. It works on the same principle, but is based on the *switchTimed* behavior instead of *switch*. *switchTtimed* was introduced in Section 8.2.4 and differs from the standard switch in that it is aware of the exact time of switching. This information is used to start the new behavior at the exact time of switching instead of the approximate time of the current step. The *switchWait1stTimed* is defined as:

```
switchWait1stTimed :: SF a (b, Event (c, DTime)) --starting SF
    -> SF (a,b) (Event d) -- SF that detects the event occurrence
    -> (SF a (b, Event (c, DTime)) -> d -> Maybe (SF a b))
       -- if a new SF is given the following happens:
       -- the SF will be run AFTER the 1st SF triggers an event
    -> SF a b
```

Listing 8.4: Adjoining Transition Behavior: switchWait1stTimed

8.4.3 Blending Transition

The blending transition is probably the most useful of the transition function. An implementation that follows the idea presented in the foundation section can be created. In the implemented approach, the transition interval length is explicitly set by the developer a priori. Although dynamically determined transition periods could be implemented easily, following methods like those used in *switchWait1st*, such an approach only make sense in cases where the user is controlling more factors through

their interaction. In those cases, the developer can achieve the same results with other functionalities in more straightforward ways. Two blending switches have been created, correlating to the external and internal switching functionalities, *kswitch* and *switch*.

The *switchTtimed* SF implements an internal switch based version of the blending transition.

```
switchTtimed :: SF a (b, Event c) -- start SF
    -> (c -> SF a b) -- SF generating function
    -> SF (b, b) b -- Interpolation kernel
        -- takes both outputs and gives the weighted final output
    -> Time -- length of the transition period
    -> SF a b
```

Listing 8.5: Internal Event Blending Transition Behavior: switchTtimed

The initial SF and SF generation function are just as in the original *switch*. Two additional parameters are required for the blending functionality. The length of the blending interval is provided as an argument and the interpolation kernel must be provided. As can be seen in the definition, a decision to use an SF was made and is discussed below. After the switching event, both SFs are run for the length of the blending interval given. The output results of each are given to the blending SF. After the blending period is over, the *switchTtimed* switches sole into the new SF, as in the original switch.

The *kswitchTtimed* SF follows the same pattern, but for the external event generation paradigm.

```
kswitchTtimed :: SF a b -- start SF
   -> SF (a,b) (Event c) -- SF that detects the event occurrence
   -> (c -> SF a b) -- SF to transition to
   -> SF (b, b) b -- interpolating kernel
       -- takes both outputs and gives the weighted final output
   -> Time -- length of the transition period
   -> SF a b
```

Listing 8.6: External Event Blending Transition Behavior: kswitchTtimed

It differs from the original *kSwitch* on an additional point. The generation function in the kSwitch was based on the premise of a "call with continuation" functionality. In the case of *kswitchTtimed* this does not make sense, as the behavior should definitely change. Instead, the basis here is the external nature of the event occurrence. Therefore, the generating function is changed to simple mirror that of the standard *switch*.

The keyframe behavior presented in Section 8.3.2 also depended on interpolation for time based functionality. In that behavior, a simple function was used to provide polymorphic interpolation kernels to a single behavior. However, the differences in the needs of the interpolation discussed in the theoretical section above, led to a different implementation approach for the blending transitions. Instead of a simple function which receives a percentage of the time of the interval, the blending kernels have been implemented with an SF themselves. Although the implementation is slightly more difficult, the use of a SF for the blending functionality instead of a simple Haskell

function brings two benefits. The most obvious is that the blending function is time dependent and should be implemented as such, i.e. a continuous time behavior. The unfortunate part of this is, that two separate places have to coordinate over the time: the time given to the switch and the SF has to interpolate over the same time. The reason that a continuous time behavior is more important is that the algorithms tend to be much more complex (linear interpolation is common for keyframing). The other benefit is though the use of an SF, the kernel can be history aware. This means that the blending function can take into account what happens during the entire blending. A Haskell function would not be able to do this. Many of the advanced kernels require such abilities, but are beyond the scope of this work to discuss in depth.

8.4.4 Blending Adjoin Transition

The final transition possibility is the blending version of the adjoin transition. This transition is the most difficult of the transitions to support. Although the visual representation of this in Figure 8.9 makes this look simple, this is only the case when enough external knowledge of the behaviors is available. Without a priori knowledge it is difficult to handle this case properly. The issue with implementing it for a general case is not knowing how long the blending will occur, a critical part of the interpolation method. In the special case of a cyclic avatar behavior, there is no issue, as there is a definite, predefined end to the behavior that is always near. Unfortunately, the author is not aware of any algorithms that can be used in the case where the end of the behavior is not known, or can be calculated, a priori. Even when the end of the transition period can be provided, another potential issue is that the time until the actual switch could be extremely long or short; either case could invalidate the method of blending. For instance, if the change is very short, even the best asymptotic functions will result in a near linear transition. When the values are not also near, this could be nearly equivalent to an instantaneous jump.

The second major issue with the blending adjoin is even more troubling: how to know what the starting value of the second SF will be. A number of issues are involved with determining the starting value. For some special cases, the starting value is either known a priori by the designer. In other cases it is easily calculated by starting the SF with $\delta t = 0$. However, this assumes a few simplifying conditions are true. Among these issues is the assumption that the creation of the new SF isn't dependent on the final output value of the initial SF, which is part of the standard SF generation function. If this is the case, then no adjoin is possible. The other question is what value to give to the SF as the initial input value. If a valid input can be given and what it is, is highly dependent on the actual functionality of the new behavior. The only case where the adjoining transition can function is where the input that is present at the time of the transitioning event is valid for the new function also. In many cases this is doubtful, as the input is unlikely to be the same value as the one at the moment of final switching.

Given the conditions that must hold to make this blending adjoin practical, a single functionality is provided for that case. This is a solution for the special case of a

cyclic behavior that the cycle time is known a priori. The *switchBTWait1st* behavior implements this functionality as an extension of the *switchTWait1st* to include the timing information. Its definition is:

```
switchBTWait1st :: SF a (b, Event d) -- starting SF
     --returns an event when passes the point
     -- where it can switch tagged with time delta
     -- to the switching point
   -> SF (a,b) (Event c) -- SF that detect the event occurrence
   -> (SF a (b, Event d) -> c -> Maybe (SF a b))
      -- if a new SF is given the following happens:
      -- the SF will be run AFTER the 1st SF triggers an event
   -> (Time -> SF (b,b) b) -- interpolation kernel builder,
         -- recieves the time until the end of the next cycle
   -> Time -- length of time that the cycle of the initial SF takes.
   -> SF a b
```

Listing 8.7: External Event Blending Adjoin Transition Behavior

Two extensions to the original *switchTWait1st* are made. The length of time that one complete cycle takes is required. The second change is that the interpolation kernel is changed to be a function with creates the SF and gets the delta time from the system as its sole argument. This has to be done, since the actual interval is only known at the time of the transitioning event. It is calculated based on the current time and the time a cycle takes.

8.5 Reduction of Programming Difficulties

The final set of extensions that are part of the FRVR system deal with issues of usability of the system. One of the motivating factors to this dissertation was in finding a method that better supported the developer of what we have termed DIVEs. The method developed on the basis of Functional Reactive Programming addresses the core nature of the design space of DIVEs. However, due to the implementation of the system, the usability in terms of ease of programming is a point that could be contested.

In the design and implementation of the FRVR system, various steps have already been taken to ease the usage of two disparate programming languages that are from two different programming paradigms. For instance, the design of Haskell math module for computer graphics specific functionality was created to mirror the library that was chosen for the C++ implementation of the data exchange system.

In this section a number of additional developments that aimed at reducing the programming difficulties involved in the FRVR system are presented. In the first section the different wrappers of complex Haskell functionalities that were developed are presented. These wrappers are designed to eliminate much of the need for the user to understand complex functionalities like the IO Monad or how the memory management and marshalling of values are performed. The second section introduces work that was done under guidance of the author on visually programming of Arrows and Yampa.

8.5.1 Haskell Functionality Wrappers

Reducing the needed access of the developer to the inner workings of FRVR is one of the directions explored in order to make the FRVR system more usable, particularly by a non Haskell expert community. A main focus in this is to remove most of the need for the developer to deal with the more complex components of the Haskell language and to deal with system level access. The inner workings of all of these functionalities are neither critical nor difficult for those initiated in Haskell and its Foreign Function Interface (FFI). For that reason the developed tools are only briefly presented.

There are three areas of FRVR programming on the Haskell side that can be supported by the creation of special functions. As discussed in Section 7.3.3, the user of FRVR has to be in control of the external loop. However, those functionalities that would ideally be completely hidden from the developer can be reduced to simple calls for the developer to simply place in their expanded loop. Likewise, the calls to the data exchange system can be completely encapsulated. This removes many programming difficulties. Finally, the data exchange system calls are further encapsulated to allow them to be used in pure Haskell, as if they did not involve low level system calls. This enables easy usage of the data exchange inside of the simulation environment, i.e. avoiding the threading of all input from the sense function and output to the actuate function. This usage and its repercussions were discussed in Section 7.6.

Although the developer requires access to the control loop and has to create the main function that starts the simulation kernel, all of the low level functionalities that are FRVR specific are wrapped into easy to use function calls. Getting the time delta, for real-time simulation at least, is reduced to a single call where the details of how it is performed are hidden (the boost time libraries are used via the FFI to provide multi-platform support). The other major set of calls is those to handle the multi-threaded environment across the data exchange system. The developer need only call two functions: *waitOnVRFrame* to wait on the VR side to prepare the inputs to the next frame and *yield2VR* after finishing the write of all outputs to allow the VR side to start again. Using these, the Haskell thread is put to sleep via low-level calls and wakes again when the VR side signals it is time to simulate the next step.

The retrieval and sending of data on the data exchange system requires use of the IO Monad in Haskell. All of the function calls involve the use of both the Foreign Function Interface and of data marshalling functions. On top of this is the memory management that is involved with sending values and also in retrieving the multiple values that make up the basic types, e.g. Quat is made up of four floating point values. Functions that wrap these difficulties, so that the developer need only to make a single call, have been created. For example, the call *writeQuat* is responsible for taking the Math type class Quat writing its value to a specified place in the data exchange. The place is defined by a String type. The function must manage the memory to convert the Quat to an array of floating point values (writing more than simple values via the FFI is difficult) that has to be allocated. The string also has to be converted to a type that is compatible, in this case a C defined Char pointer to an array. The call via the FFI

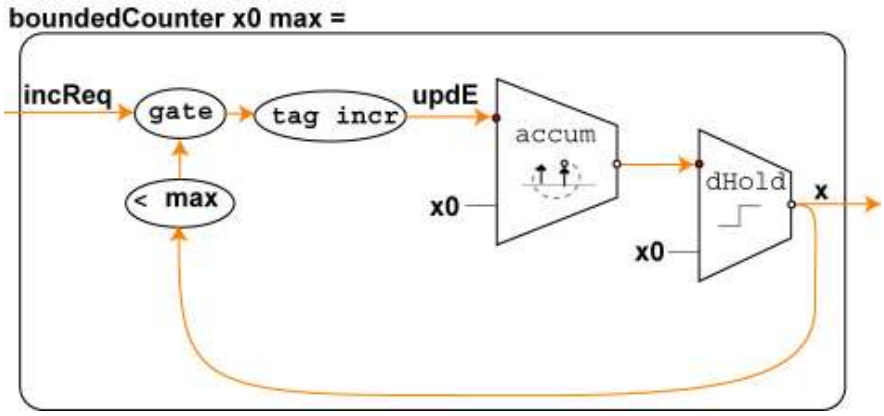


Figure 8.10: A diagram Courtney in his dissertation used to illustrate how to build a counter in his Yampa system [Cou04]. The ovals represented Haskell functions and the boxes SFs. (Used with Author Permission)

to the actual function that inserts the data in the data exchange is then called. Finally, the memory that was allocated for the conversion of the quaternion and the char array must be deallocated. This is all encapsulated in a single call using the Haskell types that are used in the simulation.

The interface created in the wrapping of those functions works perfectly in the context of the sense and actuate functions as part of the outside kernel loop. This is due to the fact that they are IO functions and that all of the FFI calls "taint" the caller as part of the IO Monad. However, their use inside the simulation is complicated by the need to declare the function as "safe." While this is a single call, its usage in the context of the simulation is somewhat tricky and requires understanding how to use point-free notation of Haskell functions (Section A.3.2 discusses point-free notation in contest of Arrows). Since the number of different functions is limited, simple wrapper functions have been created for these calls. This allows the developer to make a single easy call to get/send values from/to the data exchange. This interface is however somewhat dangerous to use, as any errors that would occur will cause the program to crash.

8.5.2 Visual Programming

One of the goals of the development of a new method for supporting the creation of Dynamic, Interactive Virtual Environments was ease their development. While we have seen that the FRP paradigm is well suited to the hybrid nature of the DIVE programming, the Yampa system is not optimal in respect to ease of programming. Not only is it written in a different language than normally used in VR, it uses complex extensions that are, even at a conceptual level, difficult to fully grasp. In the authors' experience, learning how Yampa worked was difficult, particularly given the context of a new language, Haskell, and the complex extension of Arrows.

A possible solution to this issue can be found in the descriptive parts of the FRP research. In the early portions of Courtney's dissertation [Cou04], he uses several

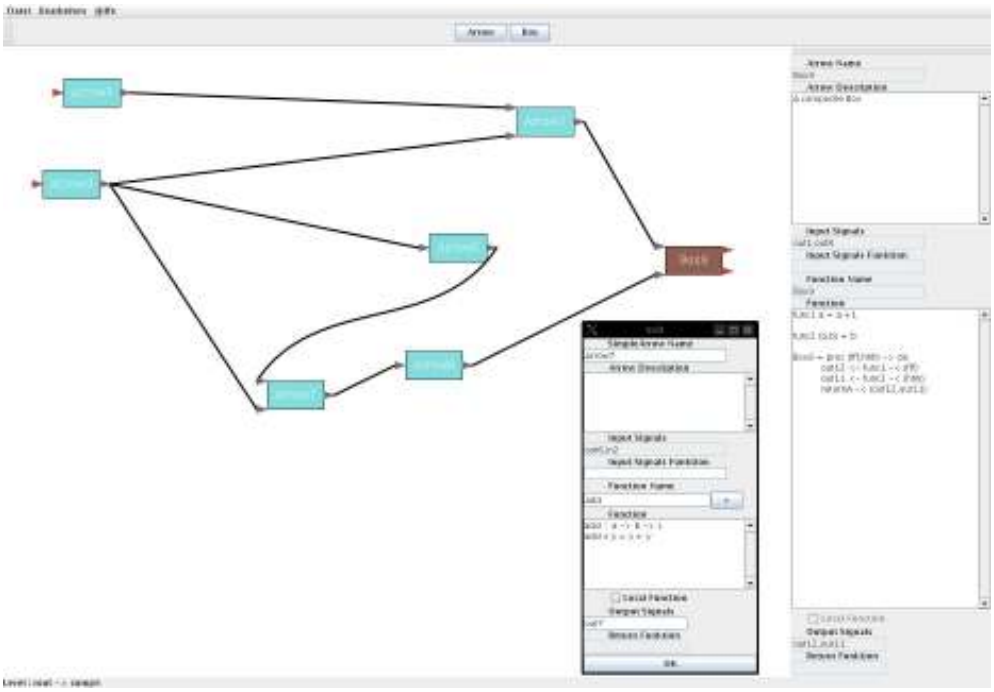
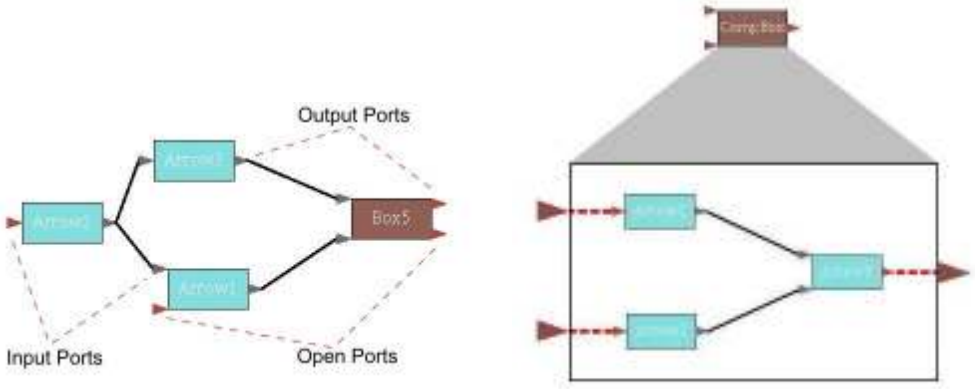


Figure 8.11: The developed Visual Programming Environment in usage. The user develops the simulation environment at the level of connecting SFs. SFs are programmed either by composition of other SFs or by lifting a Haskell program specified in a special dialog.

diagrams to illustrate how a few basic functionalities can be created. Figure 8.10 shows one of the diagrams from Courtney. Similar diagrams are used to explain the Arrow extensions on the Haskell webpage [has]. Figure A.2 recreates these diagrams. After attempting to understand the accompanying code to the diagram, the idea to use the diagrams in learning Arrows and Yampa was followed. Drawing diagrams of how the program should function turned out to be a method that worked well in early developments. Part of the reason for the success of the technique is that the translation of a diagram, such as that in Figure 8.10, to actual Yampa code is straight forward.

Based on these experiences and observations, the idea to apply visual programming to the programming of Yampa based code was considered. A German Diploma thesis to this theme was performed by Piotr Szal [Sza07], under the guidance of the author and his advisor. The results of that work are a Visual Programming Environment that can be used either for straight Arrow code or for Yampa programming. The development of the resultant system is summarized in Appendix B.

The Visual Programming Environment (VPE) enables the user to program SFs as a data flow system. Figure 8.11 shows the VPE in use. A FRVR simulation environment can be programmed using this interface. The boxes seen on the interface space are each an SF. The outputs and inputs are connected to form the program data flow. Each of the boxes is an SF of one of two types, either a lifted Haskell function or a “compound”



(a) Ports of Boxes (b) Internal Environment of a Compound Box

Figure 8.12: Basics Components of the VPE

function that is composed of other SFs. The later is demonstrated in Figure 8.12(b), where the orange box's behavior is created by the connection of three other SFs. SFs that are created via the lifting function (*arr*), require Haskell programming, for which a special editor is provided. The connecting of SF inputs and outputs occurs via the typical visual programming method of drawing connections between the inputs and outputs of different objects. Figure 8.12(a) demonstrates this. Even the loops possible in FRP code are handled.

The generation of code from the diagram proceeds through a simple export. The system can generate Yampa SFs or can alternatively produce pure Arrow code. The generated code can then be used in FRVR programs as usual. The program stubs generated are modules that can then be used as behaviors in the complete simulation or imported into other behaviors. The VPE allows the saving and loading of components and saved behaviors.

Although the VPE shown is still somewhat rough, it shows how this approach can be achieved. The programming of behaviors, particularly for beginners, is eased through its use. Using the VPE it is not necessary to understanding the Arrow syntax or even the Arrows concept. The developer can instead focus on how the program should work, instead of on specialized syntax and complex mathematical principles. The data-flow approach matches extremely well to the Arrow concept. The layered approach of building up complex behaviors is also helpful for all developers. The generated code is in Arrow form and reflects the diagram version of the code, so that debugging is eased. This also helps developers learn how the underlying code works. The VPE is a tool that helps particularly in the learning phase and for those that have little programming experience.

Chapter 9

Examples

To demonstrate the usage and abilities of the FRVR system, examples built with the system are presented here. The examples shown are chosen to highlight the development of Dynamic, Interactive Virtual Environments with FRVR. As such, they exemplify the different approaches that can be taken to create the multitude of identified Dynamics, Interactive Dynamics, and Dynamic Interactions.

The examples presented build in complexity as the chapter progresses. The first section details how a complete implementation of a trivial dynamic, including its display via the VR Juggler and OpenSceneGraph systems. Because the "system" aspects remain very similar for the more advanced systems, the remainder of the examples focus exclusively on the Dynamics, Interactive Dynamics, or Dynamic Interactions. Only in instances that require significantly different work on the VR side are these portions explained.

The first example presented in Section 9.1 builds up a complete FRVR simulation, describing all of the components required for programming a DIVE with FRVR. The dynamic developed is a trivial forward motion. Following this introduction, a set of examples based on simulating dynamics and pseudo dynamics are presented. These examples build from a simple dynamics of a pendulum to more complex concepts and are found in Section 9.2. This progression of examples also introduces many of the concepts of FRP and FRVR, including the addition of interaction. Section 9.3 introduces a time delayed behavior that demonstrates the combination of a number of functionalities together, including the usage of classical animation techniques. Section 9.4 introduces an implementation of the classical steering behavior method, Boids. This is largely a non-dynamics based system used widely in VR and computer games. Finally, Section 9.5 describes a complete DIVE developed using the FRVR system.

9.1 A First Dynamic

The creation of a simple dynamic using the FRVR system is presented in this section. The dynamic is explicitly chosen to be trivial, a simple box that moves with a constant velocity. This simple step is classical for incremental implementation of dynamics in VR and is also easy to follow. More importantly, the example highlights all components that must be developed for this simple dynamic to appear in an environment.

The development of a DIVE in FRVR begins with a few initial considerations that have already been brought up in Chapter 7, where FRVR was developed. The most fundamental decision is ownership of the value. That is, whether the FRP side controls the value or the VR side. In this example, we will decide that the FRP side controls the value. In this example, the dynamic is simply displayed; the VR side only contributes the rendering of the scene in this application. If the VR side is in control of the value, it must be passed in both directions and the SF's input must be that value threaded through the simulation environment. The next most fundamental decision is whether the simulation environment will be programmed in a pure way or not. In this case, there is no input to the simulation and only the output of the moving box's position.

VR Programming

The programming of the application happens in two places, the VR application and the Haskell/FRP application. For the VR application, an implementation in VR Juggler is shown. On the VR side a few steps have to be taken. Obviously, an object that is to be moved must be created. This involves standard scene graph operations. The code developed is based on an OpenSceneGraph (OSG) implementation. As the object is to be moved, it must be placed under a transformation node. This is part of the setup. The other part of the setup is establishing the values that will be exchanged. This has to be done in the setup stage, before the creation of the simulation thread, due to the simplistic data exchange system created (the memory must be allocated before the thread fork so that the memory can be accessed by both systems). We will name the object "box," to reflect what it is. As the only movement of this object is its position, a *gmtl::Point3f* is used to represent it. This is inserted in the data exchange system. At this point, the simulation can be initialized. The additions to the standard *initScene* function required are:

```
// Add an object to the scene to be dynamic
mBoxXform = new osg::MatrixTransform();
osg::ref_ptr<osg::Node> box = osgDB::readNodeFile("axis.osg");
mBoxXform->addChild(box.get());
mNavTrans->addChild(mBoxXform.get());//std OSG app navigateable SG tree

// insert the data into the data exchange
gmtl::Point3f box_pos;
insertPositionData(std::string("box"), box_pos);

FRVR_mixin::initScene();
```

Listing 9.1: VR Juggler/OSG Initialization of a Simple Box Dynamic

The second part of the VR Juggler application side of the code is the actual graphics loop update. This is implemented in the *preFrame()* function. Since there is not interaction in the system, the application update consists only of the running of the simulation, querying of the results, and updating the SG. Running the simulation is trivial with the support of FRVR, requiring only a single call. The call is internally blocking, so after returning the values can be immediately retrieved. This is a single call to the data exchange system, but an allocated value must be provided. After retrieving the new position, the update of the SG node must occur. The transformation node that was defined previously was stored in a member variable, so that it could be accessed again here. A simple call sets the new position by setting a new Matrix in the Transformation node. The code is complicated only by the difference in the math types between the VR Juggler system (which uses the semi-external GMTL library) and that of OSG's internal math type. The code to handle this appears as:

```
//Have FRVR do it's calculations now
FRVR_mixin::preFrame();

gmtl::Point3f box_pos;
 // The the data for the Pendulum
retrievePositionData(std::string("box"), box_pos);

osg::Matrixd boxmat;
boxmat.setTrans(box_pos[0], box_pos[1], box_pos[2]);
mBoxXform->setMatrix(boxmat);
```

Listing 9.2: Additions to VR Juggler/OSG Application Update for a Simple Box Dynamic in the Frame Loop

That completes the work that must be done on the VR Juggler side.

Haskell/FRP Programming

On the Haskell/FRP side there are again two components of the code to be created. The Yampa simulation kernel must be set up in a Haskell function. This function must be externally exported via the Foreign Function Interface (FFI). This is necessary so that the external VR (C/C++ programming language) side can start this part of the application when forking the simulation thread. The function need only perform a few simple calls. It makes a single call to the simulation kernel. The simulation kernel is a loop that only ends from an internal result provided by the simulation. We will use the standard Yampa simulation kernel here, reactimate. It requires four arguments: the value to initialize the environment with, the sense function, the actuate function, and the simulation environment (an SF). This simple application has no input, so a *()* is provided. Since there is no input, the sense function only retrieves the time delta. The actuate function is dependent on the output of the system. We know that only a single position value is required; the developed Haskell Math library defines this with the *Point3* type.

Therefore, the code will look like this:

```
runFRVRHaskell :: IO ()
runFRVRHaskell = do
                 waitOnVRFrame --wait for first frame
                 reactimate () -- init with no value
                            sense -- sense function
                            actuateBox -- actuate function
                            (movingBox (Point3 0.0 1.0 0.0)
                                       (Vector3 0.0 0.0 (-1.0))
                                       (0.1)) -- creates SF environment
                 return ()
                 where sense :: Bool -> IO (DTime, Maybe ())
                       sense _ = do
                                 waitOnVRFrame
                                 dtime <- getElapsedTime
                                 return (dtime, Nothing)

actuateBox :: Bool -> Point3f -> IO Bool
actuateBox _  box = do
        _ <- writePosition "box" box
        _ <- withCString "RunHaskell" clearEventInBB
        _ <- withCString "FRVRComplete" setEventInBB
        _ <- yield2VR
        return (False) -- success
```

Listing 9.3: Haskell Simulation Setup Code

Finally, the actual simulation must be created. The function that creates the simulation environment can already be seen in the code above. To make the code more interesting and because the FRP side is in control of the value, an initializing position is given as a parameter. In this function the desired behavior is created. The behavior should be a box moving with constant velocity. The direction of travel is defined by a three value vector, *Vector3*, and also provided as an argument to the constructing function. The direction of travel is defined above as the $-z$ axis (forward in the OpenGL based display). Given a direction of travel the velocity needs to be known also and is additionally provided as a parameter.

Provided with all the basis information, the implementation of the actual dynamic is very simple. The motion of an object moving with constant velocity (in frictionless space) is defined in physics as:

$$p = p_0 + \int v \, dt \tag{9.1}$$

This translates directly to the behavior code:

```
movingBox :: Point3f -> -- ^ initial position
             Vector3f -> -- ^ direction of travel
             Float -> -- ^ velocity
             SF () Point3f
movingBox iPos iDir vel = proc _ -> do
         pos <- arr (translatePoint iPos) <<< integral -< vel ^* iDir
         returnA -< pos
```

Listing 9.4: Creation of a Behavior of a Box Moving at Constant Velocity

Here we see the similarity of the code. Haskell based code including arrows always takes an input. The behavior is defined here to take a *()*, the NULL value of Haskell. The input to the integral is the velocity in 3D space via a vector. The result of the integration is a Vector3f type. Since a Point3f is desired as the output and the type of the initial value, the special function *translatePoint* is used. Here we see the use of a Haskell specialty, "currying" the function with one of the two values and then passing it as a single valued function to the *arr* (lifting) function.

Putting It Together

Haskell's FFI combined with the developed data exchange system make the connection of the two systems easy. The data exchange system makes the connection automatically for the end developer in all points, but one. This has been spoken of already in the VR section above. The initial call from the VR side goes to a function that is defined in Haskell and must be exported by the FFI. This call though, can be simply copied from the examples, so making new applications is easy.

The complication in getting an executable is compiling the different components. FRVR to date uses the GHC compiler for the Haskell side. VR Juggler comes with example applications that be used as a basis and tools that help application developers with compilation requirements. Compiling the each individually is not difficult, but the connection of the two at the time of linking everything together can be challenging. To assist this, FRVR provides a special "Makefile" to be included application Makefiles that sets up all required compilation flags, macros for compiling, and dependency tools. Makefiles are part of a build system commonly used in the Unix OS area and in many VR systems, including VR Juggler.

For the example above, the preparation of the compilation environment involves creation of a Makefile to control the build process. Using the examples provided in FRVR, this is generally just a process of changing the file names of the sources to be compiled. Listing 9.5 below shows the critical portions of the Makefile. The name of the executable to be created must be provided. All C++ source files are listed in the SRCS variable and are automatically compiled. The special FRVR Makefile has to be included (when used with VR Juggler must be placed after the includes of VR Juggler's included Makefiles). Similar to the C++ sources, the Haskell files to be compiled are listed, but in the form of compiled objects (the .o extension indicates a compiled file containing the code of that module). The file will automatically be found and compiled by the GHC compiler.

```
APP_NAME=       osgNav
SRCS=           OsgNav.cpp main.cpp

FRVR_BASE_DIR ?= ../../FRVR
include ${FRVR_BASE_DIR}/frvr.app.mk

HASKELL_OBJS += FRVR_main.o
OBJS += ${HASKELL_OBJS} FRVR_main_stub.o
```

Listing 9.5: Critical Components of Build System

The combination of the two systems requires a few things to be done and watched for. Here, the FRVR_main.hs file is compiled to create the require object file. Since it is the file containing the export of the main function that the VR system calls to start the Haskell thread, a C interface to that function is created in a generated interface file, in this case FRVR_main_stub.h and FRVR_main_stub.c. These files have to be compiled also, but the C/C++ compiler. Because of this, the list of OBJS has to be extended to include the compilation of this stub. The linking of all the objects together is performed by the GHC compiler, with the necessary call structure transparently controlled by the FRVR Makefile.

The only additional point that needs to be addressed is the actual call of the *make* program. Because there are cross dependencies (the stub specifically), the make process does not always work correctly on the first call or keep track of dependency changes between systems. Two special calls to the make system can take care of these issues. 'make -i' tells the make system to ignore problems and continue compiling other objects. This can be used on initial compilation to overcome dependency issues. The special 'make clean' command can be used to clean all object files when dependency updates are not recognized and force new compilation. In the general case, a simple 'make' is adequate.

9.2 Dynamics and Pseudo Dynamics

A series of examples based on classical dynamics are highlighted in this section. As with the example of the last section, the implementation of forward dynamics and mathematical based behaviors plays to the strengths of the FRP paradigm. This class of dynamics is well understood from classical physics and mechanics and is well defined. The examples presented here build up in complexity, from simple dynamics to relatively complex, interactive, pseudo dynamics.

The first example is a trivial simulation of a body that launched in an environment with gravity. It provides a gentle introduction to how the simulation works and is important, as it is a common starting point for dynamics when considering implementing such things. After this, a number of examples that illustrate different usages of FRVR are developed on hand a pendulum simulation. Initial pendulum examples demonstrate how to implement the rotational motion of a pendulum and how interaction can be incorporated in a behavior. A further example extends the pendulum to advanced simulations, the Newton's Cradle example and a two-point pendulum.

9.2.1 Simple Dynamics

The first dynamics many seek to create in a VE is having an object that is affected by gravity flying through the environment. In the previous section a complete example was shown of an object moving with constant velocity. This section focuses on the introduction of a dynamic that is slightly more complicated. Whether it is a reproduction

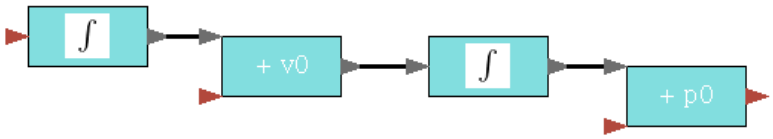


Figure 9.1: The launching of a object programmed in the Visual Programming Environment. The same function as in Listing 9.6 is created.

of the classical artillery game (like the Scorched Earth game) [Bar] or just throwing an object, the goal of this is to simulate real world physics where the object moves from some impulse velocity and is affected by gravity. The starting point of implementing the dynamics is the physical laws that we want to hold:

$$v = v_0 + \int a \, dt \tag{9.2}$$

$$p = p_0 + \int v \, dt \tag{9.3}$$

where v is the velocity, a the acceleration, and p the position.

At this point we will just ignore where the impulse to the object comes from, assuming it is a "magical" source. This keeps the example simple and is valid in a synthetic world. An object moving with a velocity in a frictionless environment was developed in the last section. Equation 9.3 is the same as used in that section. However, now we are adding gravity; therefore, the velocity now changes constantly as per Equation 9.2. The acceleration due to gravity is simply a constant $-9.8m/s$. The formulas of Equation 9.2 and 9.3 are directly transferable to FRVR/Yampa and produce the following code:

```
g :: Vector3
g = Vector3 0 (-9.8) 0

flying_la_vache :: Vector3f -> SF () Point3f
flying_la_vache v0 = proc _ -> do
    velocity <- arr (v0 + ) <<<
                integral -< g
    (Vector3 x y z) <- arr (p0 + ) <<<
                       integral -< velocity
    returnA -< (Point3 x y z)
    where p0 = (Vector3 0 0 0)
```

Listing 9.6: Simulation of a Launched Object.

This simulation code can be used as the simulation code in the same code base as developed in the previous section. The same code can be created using the Visual Programming Environment presented in Section 8.5.2. The necessary "code" diagram can be seen in Figure 9.1.

9.2.2 Pendulum

In this section, the simulation of a simple pendulum is presented. Two different implementations of the pendulum will be presented. The first implementation is done in the straight forward manner, using a simplified mathematical model of the pendulum. Although this implementation approach is valid, its usability is limited by simulation factors. Those limitations are explained and an alternative implementation method that overcomes these limitations is introduced in the second part. This example provides a number of important insights into programming more complex behaviors with an easily understood system. It is also the foundation of the examples that follow. Those build on the pendulum to show how more complex behaviors can be built.

Simple Dynamics Pendulum

The dynamics of a pendulum are more complicated than those of the trajectory example in the previous section. From physics, we know that the rotation momentum of a body is more complicated to calculate. However, for our usage some simplifying assumptions can be made. First we restrict our model to a rigid arm pendulum and reduced its freedom of movement to a single axis. The simplified equations for such a frictionless pendulum are:

$$\alpha = \frac{-g}{L} \sin\theta \tag{9.4}$$

$$\omega = \int \alpha \, dt \tag{9.5}$$

$$\theta = \int \phi \, dt \tag{9.6}$$

where L is the length of pendulum and g is the force of gravity.

The implementation of these formulas into code is similar to the dynamics code above. There are however a few differences to the formula. α is angular acceleration of the pendulum and is dependent on the angle of the pendulum. The angular velocity (ω) is derived from the angular acceleration, and the angle of the pendulum from the angular velocity. Here we see the first difference in the defining algorithms to the previous dynamics, as there is a loop in the dependencies. The other difference comes when we consider the implementation of these algorithms.

The implementation is complicated by the fact that we are now dealing with orientations as the basic unit instead of position. Equation 9.4 assumes the angle θ is the angle of the mass with respect to gravity. Since we have assumed a single degree of freedom, the angle on that axis suffices and we only need to calculate the angle between it and gravity. The second, and more important, complication is the integration of the angle components. This is conceptually the same as velocity and acceleration above, but, in implementation terms, this is not always simple depending on the representation chosen. The standard method for representing angles in such situations is quaternions [Sho85].

Using quaternions for the rotations and operations on them in terms of physically based modelling is a topic of interest for many because of their properties. Good references on using quaternions for physical simulation can be found in [Hof05] and [WBK01]. Quaternions have the benefit of permitting easy calculation of integration without the numerical drift inherit in the traditional method using matrices. The formula that we need is defined by:

$$q(t) = \frac{1}{2}\omega(t)q(t),\ where \tag{9.7}$$

$$\omega(t) = [cos(\frac{\theta}{2}),\ sin(\frac{\theta}{2})\overrightarrow{u}] \tag{9.8}$$

In Equation 9.8, θ is the angle of rotation defined around the axis defined by the three valued vector $\overrightarrow{u}$. This is set in the form of a 4 value vector that is the same as a quaternion and can therefore be multiplied. Equation 9.7 can be used in our integration to get the angle θ when it is in quaternion form. The only additional step that must be taken is to maintain a unit quaternion during integration, as only the unit quaternion is a valid representation of an orientation and the integration does not return a unit quaternion.

Based on the Equations 9.4, 9.5, 9.6 and 9.7, the simulation code for a pendulum is given below. The SF is assumed to be in control of the simulation and only the angle of the single axis pivot is calculated. The direction of gravity is assumed to be that of an OpenGL oriented world. The code displayed in Listing 9.7 can be run as the simulation kernel in a system very similar to the complete system presented in Section 9.1. The only difference is that, instead of a position, a quaternion value must be used.

```
g, l :: Float
g = -9.8
l = 10.0

-- this function calculates the angular acceleration,
-- based on the angle the pendulum has to gravity.
calcAngularAccel :: Quatf -> Float
calcAngularAccel ang |(vectorLength ang) == 0 = 0.0
                     |otherwise               = -g *
           sin ( (angle (toAxisAngle (QuatRot ang)))) / l

-- constructs a SF which does a simple pendulum physics
-- quat input is the initial position of the pendulum
-- the function assumes that it controls the position
testPendulum :: Quatf -> SF () Quatf
testPendulum initial_angle = proc dtime -> do
  let -- helper function for angle to gravity calc
      -- assumes the -Y axis for OGL coordinates
      angle2gravity rot' = rot' / (Quat 0  0 (-1) 0)

  rec --recursive since rot is used here and set at the end
    --calculate the angular acceleration
      angular_accel <- arr calcAngularAccel -<
                                (angle2gravity rot)

      -- the angular vel calculated from angular accel
      angular_velocity <- imIntegral 0  -< angular_accel
```

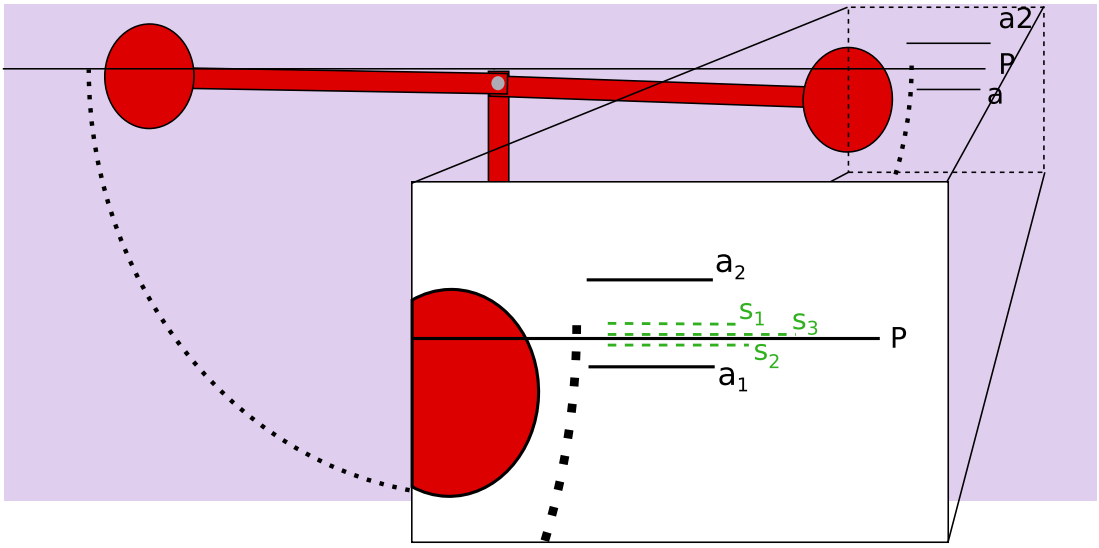


Figure 9.2: An illustration of how the transitioning pendulum approach improves the sampling of the time of transition. Here the *subsampledSwitch* is demonstrated.

```
    -- perform quaternion integration
    rot <- iPre initial_angle <<<    -- the initial angle is set
       -- accumlate the angle and normalize the new quat
    arr (normalize.(initial_angle+)) <<< integral -<
       0.5 *^ ((Quat 0  0 angular_velocity  0)*rot)
returnA -< rot
```

Listing 9.7: Code for Simulating a Pendulum

This implementation of the pendulum is viable, but runs with a caveat. Here, the simulation sampling rate becomes an issue as discussed in Section 8.2. Of critical importance here is not the general sampling rate, but the moment of sampling at the peak of the pendulums arc. The root of this sampling problem was discussed in general in Section 8.2 and the pendulum in specific in Section 8.2.3. Various methods to reduce or eliminate this effect were proposed in that section and a solution based on those is presented next.

Transitioning Pendulum

The previous subsection presented an implementation of a pendulum based on the physical properties of such devices. That implementation suffers from a sampling issue related to the higher-order transition of the pendulum at the peak of its arc. A special solution was developed for FRVR for addressing this general issue in Section 8.2.3.

There are a few potential ways to address the issue. Sampling at a higher rate would ensure the time of sampling is closer to the actual moment of the peaking arc. However, it also means sampling all other points excessively. In this case only the moment of

the highest point of the arc is necessary to be sampled at the higher rate. Because the implementation already uses the higher-order equations to define the motion, we can use this to find the moment of transition. The peak of the arc of the pendulum is the moment when the angular velocity crosses the zero point, i.e. a change in signs. This means the point can be calculated. The *switchTimed* behavior could be used, but the calculation of the point of transition is quite expensive. Instead the sub-sampling switches of Section 8.2.3 can be used.

The implementation of a pendulum that takes advantage of this development is similar to that of our original pendulum. The key to this new approach is to use moments where the velocity changes direction as switching events. Since the generation of the event relies on information internal to the behavior, Yampa's internal switches are optimal. The *subsampleSwitch* and *subsampledSwitch* are both sub-sampling switches based on this internally generated event principle. On the event, these special switches re-sample the SF at a higher resolution. This mechanism increased the resolution at the moment it is necessary, while incurring only minimal extra costs. Figure 9.2 illustrates the principle in use, based on the converging switch, *subsampledSwitch*.

To generate the event, we only need to make a small addition to the existing pendulum behavior, namely the detection of the change in the sign of the velocity from the last frame to this frame. The code for the detection of the change is:

```
-- detect the sign change in the velocity
vel_sign_change <- identity -< if
          ((signum angular_velocity) /= (signum prev_vel))
          then Event (rot, angular_velocity)
          else NoEvent

-- need to keep around the velocity for comparison
prev_vel <- iPre initial_vel -< angular_velocity
```

Listing 9.8: Detection of Pendulum Velocity Transition

The second line of code is responsible for keeping track of the velocity in the previous frame. The first line of code is a Haskell function that simply tests the sign of the value. If the sign changes, the *Event* is thrown.

The creation of the behavior testPendulum now creates an Event, but we have yet to address the switching. A wrapping function that handles the switching environment has to be created. On switching, we wish to enter the same behavior, just moving the other direction. For Yampa switches, this requires a recursive call (see Appendix A.4.2). The creation function sets up an initial switch and also the function needed for creating a recursive call, i.e. the same Event extended pendulum SF is always newly generated.

Unfortunately, because of the change to the transitioning algorithm, a few additional points have to be addressed. In the original code (Listing 9.7), the starting angle was provided by an external source. With the transition approach, the angle at the moment of the event needs to be provided to the new behavior. The previous algorithm also assumed that the velocity is also 0 when the algorithm starts. Slight inaccuracies in the moment of sampling of the switch lead the value of the velocity to be not exactly 0. Leaving this as 0 could be done; however, to be complete, the velocity at the moment

of transition could be transferred to the new pendulum behavior. Additionally, this provides a value to compare to on first pass of the new behavior. That is, on the initial run of the new behavior the last velocity was generated by the previous behavior. The simulation is then extended to take an initial angle θ and an initial angular velocity ω. The Event shown above was tagged with these values.

One final issue remains. Unfortunately, the integral function that was previously used for the calculation of the angle from the angular velocity introduces a single frame delay. This is done to ensure that the system is stable, but is unacceptable in this case. In essence this means that the initial velocity is always zero and the position doesn't get affected by the incoming velocity for two frames. The solution is to change to the "immediate integral" behavior, *imIntegral*. *imIntegral* is provided by Yampa and requires initialization with the previous value, which we happen have. Developers that are using transitioning approaches need to be aware of this when designing their simulations, as Yampa assumes that the delay is acceptable in order to avoid the possibility of invalid data.

The complete code for the transitioning pendulum's behavior is:

```
testPendulum :: Quatf -> SF () Quatf
testPendulum init_angle = proc input -> do
   -- re sample the transition with the divide and
   -- conquer method, in this case halfing the
   -- dt 10 times
   ret_val <-  switchTsampled
           (transitioningPendulum (init_angle, 0)) -- init SF
           10                      -- num of sub division steps
           trans_pend                 -- function to gen new SF
           -< ()

   returnA -< ret_val
   -- helper function which generates the new pendulum SF
   -- for the next arch
   where trans_pend c = switchTsampled
                (transitioningPendulum c) 10 trans_pend

transitioningPendulum :: (Quatf, Float) ->
                         SF () (Quatf, Event (Quatf, Float))
transitioningPendulum (init_angle, init_vel) = proc _ -> do
   let angle2gravity rot' = rot' / (Quat 0  0 (-1) 0)
   rec
      ang_accel <- arr calcAngularAccel -< (angle2gravity rot)

      angular_velocity <- imIntegral init_vel  -< ang_accel

      -- changed to use imIntegral, which needs an
      -- initial value
      rot <- iPre initial_angle <<<
             arr (normalize.(init_angle+)) <<<
             imIntegral  (0.5 *^
                ((Quat 0  0 (init_vel) 0)*initial_angle)) -<
             0.5 *^ ((Quat 0  0 angular_velocity  0)*rot)

      -- detect the sign change in the velocity
      vel_sign_change <- identity -< if
                ((signum angular_velocity) /= (signum prev_vel))
                then Event (rot, angular_velocity)
                else NoEvent
```

```
        -- need to keep around the velocit for comparison
        prev_vel <- iPre initial_vel -< angular_velocity

    returnA -< (rot, vel_sign_change)
```

Listing 9.9: Simulating a Pendulum Using a Transitioning Approach.

9.2.3 Interactive Pendulum

One of the major points of interest of this work was incorporating interaction and dynamics together. The simple dynamics pendulum is extended to be interactive in this section. The example builds a form of interactive dynamics, where the manipulation causes changes to the dynamics. During manipulation, the dynamics behavior is inactive and starts anew when the manipulation ends. After release of the pendulum, i.e. the end of manipulation, the pendulum starts with a new orientation and an assumed zero velocity.

A very simplistic form of direct manipulation is used in this demonstrative example, so that the focus lies on the Interactive Dynamic nature and not on the implementation of a complex interaction. During manipulation, the angle of the pendulum is set to mirror the angle of the input device, e.g. the wand. In addition to the responsibility for delivery of the wand orientation, the VR system also needs to give queues to the simulation as to which behavior should be active. A simple approach is taken, where the manipulation behavior is active while a button is pressed. This is a fairly standard VR "mode" based interaction type, even where dynamics are not involved. For the VR Juggler based example, the *preFrame* update is modified to provide the inputs required:

```
void OsgNav::preFrame()
{
   // Get wand data
   gmtl::Matrix44f wandMatrix = mWand->getData();

   // If the user initiates interaction with the pendulum
   if ( mButton1->getData() == gadget::Digital::TOGGLE_ON)
   {
      SimpleBlackBoard::instance()->insertEvent("SelectPendulum");

      gmtl::Quatf wand_rot;
      insertQuatData(std::string("Wand") , set(wand_rot, wandMatrix) );
   }

   // If the user is interacting with the pendulum
   if ( mButton1->getData() == gadget::Digital::ON)
   {
      gmtl::Quatf wand_rot;
      insertQuatData(std::string("Wand") , set(wand_rot, wandMatrix) );
   }

   // If the user stops interacting with the pendulum
   if ( mButton1->getData() == gadget::Digital::TOGGLE_OFF)
   {
      SimpleBlackBoard::instance()->insertEvent("deSelectPendulum");
   }
```

```
    //Have FRVR do it's calculations now
    FRVR_mixin::preFrame();


     // The the data for the Pendulum
    retrieveQuatData(std::string("Pendulum"), mFRVRPendulumFolcrum);

    // insert the new rotation into the system
    osg::Matrix pendulum_matrix = mModelTrans->getMatrix();
    pendulum_matrix.makeRotate(gmtlQuat2osgQuat(mFRVRPendulumFolcrum));
    mModelTrans->setMatrix(pendulum_matrix);
}
```

Listing 9.10: VR Juggler preFrame update with user input transferred. Only the relevant portions of the code are shown.

The implementation method for the behavior can be derived from the conceptual description of the interaction. Our proposed method implies that there are two distinct behaviors, one is the dynamics of a pendulum and the other is a form of Dynamic Interaction, specifically a direct manipulation. The independence of the two behaviors is advantageous, as both can be independently developed. The dynamics, as in Section 9.2.2, already satisfy the requirements for the dynamic behavior of the pendulum. The direct manipulation method is easy to implement, provided the input of the wand orientation into the FRP system. In the simple method used here, the orientation of the pendulum is derived from the orientation of the user input device. However, the pendulum is constrained to a single axis, so only that component of the input devices orientation is used. The manipulation behavior is defined as:

```
constrainedManip :: String  -- name of manip tool orientation
                 -> ()      -- need to make this a function
                 -> Quatf -- new angle
constrainedManip toolName _ = let
     (x, y, z, w) =  getQuat toolName
     wand = Quat x y z w
     aa = (toAxisAngle (QuatRot wand))
     angle' = angle aa
     vec = axis aa
     vl = vectorLength vec
     zcomponent (Vector3 x y z) = z / vl
     constrained_wand = (toQuat (AxisAngleRot
       (AxisAngle (Vector3 0 0 1) (angle' * zcomponent vec))))
                              in constrained_wand

manipulatePendulum  :: (Float, Quatf)
         -> SF () (Quatf, Event (Float, Quatf))
manipulatePendulum (vel, init_rotation) = proc _ -> do
   let deselection_event = consumeEvent "deSelectPendulum"
   constrained_wand <- arr (constrainedManip "Wand") -< ()
   returnA -< (constrained_wand, tag deselection_event (0,
constrained_wand))
```

Listing 9.11: Simple Direct Manipulation in FRVR

This code illustrates well, not only how the manipulation is performed, but an alternative approach in cases like this. Here, we see that the major work of the behavior is not performed within the SF, but instead in the form of a Haskell function.

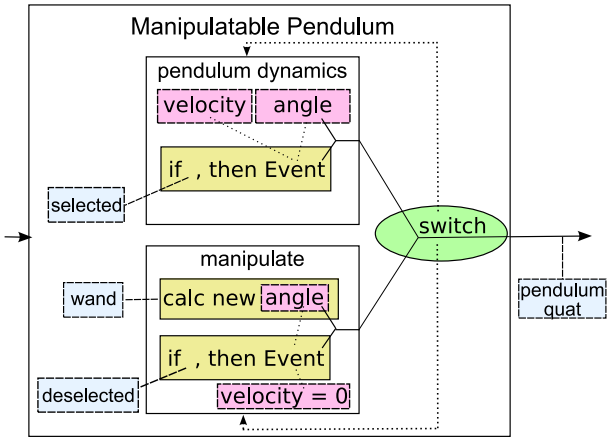


Figure 9.3: A diagram showing the conceptual architecture of the manipulate pendulum example. Either the pendulum dynamics or the manipulation functionality is active and controlled by the switch, which reacts to events thrown by the behaviors.

With both behaviors developed, the next stage is to make a single Interactive Dynamic out of the two behaviors. Which behavior should be active is determined by the state of a button. When it is pressed, the direct manipulation should be active; when it is not pressed, the dynamic should run. Unfortunately, FRP is equipped to handle reactions to events, not to states. Luckily, most VR input systems implement an event based input metaphor for just such a purpose. With this slight change in viewpoint, the active behavior is specified instead by the change in button state, i.e. on button press event and button release event. Therefore, our mode switching is now described by: button press event triggers the manipulation behavior and button release event triggers the dynamics behavior.

The Interactive Dynamic is conceptually visualized in Figure 9.3. In contrast to the switching behavior of the transitioning pendulum of Section 9.2.2, a basic switch suffices. The code for this is found in Listing 9.12 on the next page. The wrapping switch need only change between the pendulum behavior and the manipulation behavior at the right times.

```
testPendulum :: Quatf -> SF () Quatf
testPendulum init_quat = switch_dynamics (0, init_quat)
  where switch_dynamics q0 = dSwitch (simplePendulum q0) switch_interactive
        switch_interactive q0' = dSwitch (maniplatePendulum q0') switch_dynamics
```

Listing 9.12: Switching States of the Direct Manip Pendulum

9.2.4 Newton's Cradle

The Newton's Cradle example expands on the interactive pendulum simulation by incorporating interaction between multiple objects. The Newton's Cradle is a popular physics based "office toy" seen in Figure 9.4. The physical toy presents two basic

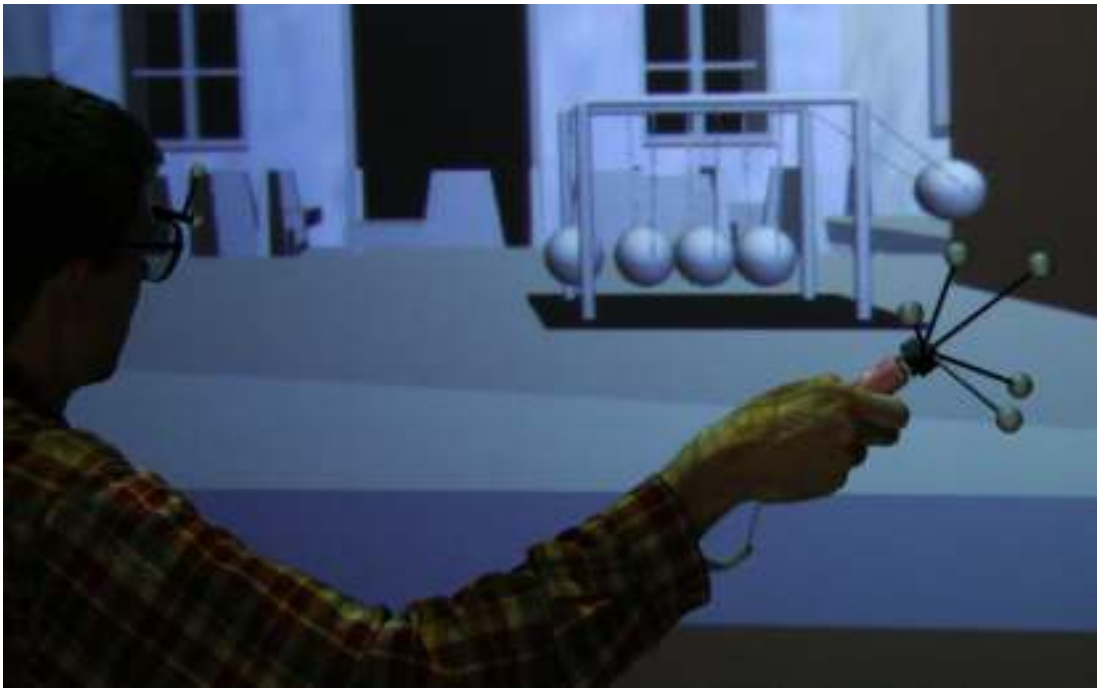


Figure 9.4: A user can be seen interacting with the Newton's Cradle. When interacting, the rotation of the wand controls the angle of the ball.

possible interactions: the user lifts one ball and releases it to start the physical dynamic or the user lifts two balls. The other balls are stationary in the middle. When the moving balls collide with the stationary balls, the kinetic energy is transferred to the ball(s) on the other side and they begin moving. The ball that just collided stops completely. Other permutations of balls set in motion, lead to the dynamics behavior of either the one or two ball motions. Disregarding friction, this repeats indefinitely.

A number of factors make this an interesting object to simulate. Multiple dynamic objects are part of one "unit," but only through interaction with each other. The object is familiar to many. It is a "playful" object that hopefully entices the user to play with it. The final point of interest is that the simulation of a Newton's Cradle using a complete physics simulation is difficult because of the internal collisions between the balls. The pseudo physics simulation that can be performed in FRVR produces satisfactory visual results, yet tractable from a computational standpoint.

Simulating the Newton's Cradle satisfactorily can be performed in a straight-forward way in FRVR. We will present a simplified one ball version here, but with interaction included. This shows how Interactive Dynamics can be both interactive within the simulated environment and with the user in FRVR without difficulty. The first observation needed for simulating the Newton's Cradle is that each ball acts as a pendulum, so we can use our previous pendulum code. The manipulation previously presented can also

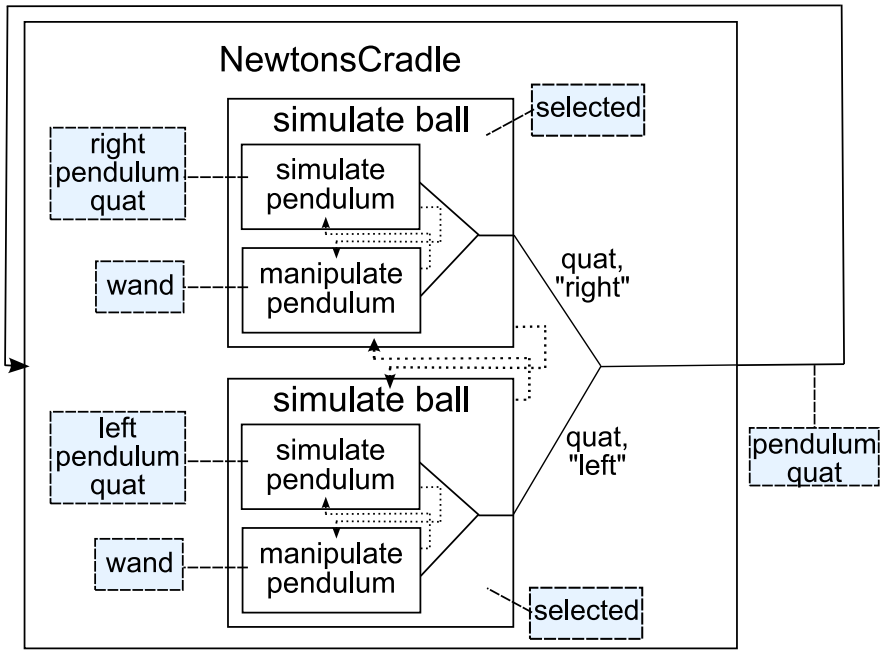


Figure 9.5: This diagram illustrates the code structure to handle the dynamics and interaction of the Newton's Cradle. Dotted lines show initialization inputs and the shaded boxes show accesses to the shared memory for data.

be used. The completion of the code to handle the interaction between the balls of the Newton's Cradle is the final component required.

Performing the complete physics is beyond the computational model necessary for realistic performance. A few simplifying assumptions can be made. The most important is that, when a ball hits the stationary balls in the middle, we can assume that the momentum is instantly transferred to the ball on the other end. Being an instantaneous interaction, an event based implementation is a good solution. Additionally, we note that only one ball is active at a time. The final simplification we will make is how to detect the collision. The three balls in the middle are always at rest in our model of the Newton's Cradle. This means the collision will occur the moment the ball reaches 0°, assuming this is the "down" orientation.

Given these assumptions, we can build a simulation where the ball at each end is independently simulated and, upon a collision event, the opposite ball is set in motion. Figure 9.5 shows the structure of the simulation. The implementation of the simulation is straight forward. The only complication is making sure the proper ball is simulated, i.e. the "right" ball or "left" ball. Since the interactive pendulum simulation of the previous section is used, that code has to be extendedly slightly to allow specification of the orientation of which ball to retrieve from the data exchange system. If we recall that this is done via the named tagging, this reduces to receiving a string for either

"rightBall" or "leftBall." that is used in the data exchange call. The behavior of the Newton's Cradle, minus the already known components, is defined as:

```
newtonsCradle :: ((Float, Quatf), String) -> SF () (Quatf, String)
newtonsCradle (inits, init_ball) =
                dSwitch (retrieveBall2Sim (inits, init_ball)) switch_sides
    where switch_sides (q0, "RightBall") = newtonsCradle (q0, "LeftBall")
          switch_sides (q0, "LeftBall")  = newtonsCradle (q0,  "RightBall")
```

9.3 Follower - A Time Delay Behavior

An important class of DIVE functionalities depends on the ability to record what happens in the world and play it back. This section shows how such functionality can be created in FRVR on hand a specific example of a time delayed behavior.

Perhaps the most important of the behaviors that rely on this is the ability to record the experience of the user and play it back later. This could be done for many reasons. Simply to allow the user to replay the experience is one facet. Even there, this could be in real-time or in the popular "bullet-time" slow motion replay method. This ability is crucial to many application areas where the user should learn from their experience. In these contexts, a review of their actions and the reactions of the system is performed. The review may require the replay be stopped, played slow, rewound, or fastfowarded. Another usage is the preparation of precanned demos, where an expert user defines a path that is later played back for demos without requiring experts to give the demo. Here, a special version of this time delayed playback is developed. In this version, the time delay is small, which makes this a worst case scenario as the potential issues due to the real-time considerations are strongest in this case.

The example here stems from a classic advanced functionality found in a number of systems. The basic concept is to detect the movement of an object and copy it. Typically, the copy cat object is offset in 3D space, i.e. exploiting the strength of the VR/SG system. The alternative of interest here is an offset of behavior in time. The idea of delaying the display of a behavior a set amount seems trivial, until one considers its implementation. For systems without any support beyond a δt, this task is daunting. If the original is a behavior of the simulation, this is easy to accomplish; the execution of a copy of the simulation can simply be started after the delay required. However, we will focus having a copy cat behavior that is a delayed version of the user's behavior, which means it is externally generated and not known a priori.

At a conceptual level, we need to "record" what occurs into some sort of stream that we can then use after a delay to create the behavior of the copy cat object. The recording of a behavior may seem trivial, but in our real-time interactive system context the variable frame update rate is again an issue. The result of this is that values recorded will not match up with the timing at playback time and the playback will run incorrectly.

Optimally, we would like to have exactly this, a continuous function built out of the observed behavior that we then just play back after some time.Unfortunately, deriving a

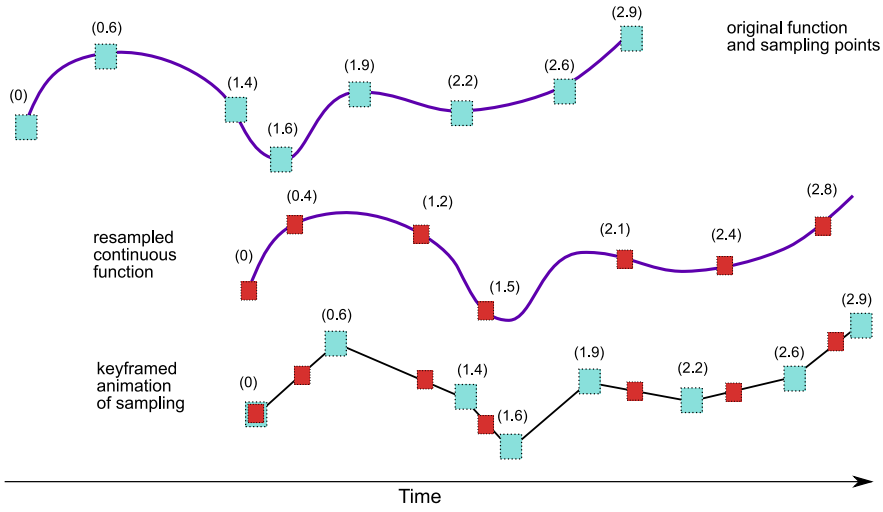


Figure 9.6: The diagram shows the concept of a delay behavior as well as two approaches to the problem. The time delayed version demonstrate both resampling of the continuous function derived from the sampled points of the original or using the sampled points as part of a keyframed animation.

continuous function from timed points is quite computationally intensive. The classical animation method of keyframing is a prime alternative, as it is designed as an alternative for precisely the problem of reducing the specification of continuous functions to key timed points. The desired functionality is conceptualized in Figure 9.6.

A keyframing behavior was developed in Section 8.3.2 making this approach seem very feasible. That behavior is created with a set of keys and timings. It simulates a continuous behavior in real-time, finding values for times between keys through interpolation. To use the keyframed method, we need to generate the keys for it to use. This requires a behavior that records a continuous behavior by sampling it. The sampling creates the keys and timing information.

Several approaches could be taken for sampling a continuous value. The easiest is to create a key for each frame of the simulation, i.e. every execution creates a key and timing info. This method has the disadvantage of always creating a lot of keys, even when the resolution is not required. Alternatively, the sampling rate could be specified. Yampa is equipped already with a *sample* behavior. It samples the input every X seconds, generating an Event when the sample is taken. Unfortunately, it does not return the actual time of sampling; therefore, we only know that the time since the previous sampling is greater than or equals to X. A more precise time stamp to the key value should be produced. Therefore, an extended sampling function is created with a signature of:

```
sampleTimeDeltaStamped :: Time -> SF a (Event (DTime, a))
```

This just a simple extension of the original *sample* behavior to additionally tag the *Event* with the actual timing of the key.

The method highlighted here can now be used for the generation of a stream of keys and timings. The results of this can be given to the *keyedAnimation* behavior for playback. This suffices for the general case of playing back a recorded behavior later. However, we desired to implement a special case, a dynamic follower with a delay short enough that the behavior that is recorded is still in process as the copy cat behavior starts. Unfortunately, the keyframing behavior implementation presented in Section 8.3.2 is not able to handle a dynamically changing list, i.e. the dynamically changing stream that will be produced by the sampling.

Fortunately, this issue of dynamic streaming can be address with a simple adaptation of the keyframe behavior. By adding the sampling functionality directly into the keyframe behavior, a stream can be created from the sampling output to the keyframe input. The resultant function is defined as:

```
follower :: RealFloat b => (a -> a -> b -> a) --interpolation kernel to use
        -> Time     -- time delay to "follow" behind
        -> Time     -- sampling rate
        -> a -- value until the values become valid
           -- note: it jumps to the first position when
           -- it starts going (after the second sampling)
        -> SF a a
```

The only complication is assuring that the keyframe algorithm requires two values at all times; that is, it requires the key of the next time in the future, so that it has values to interpolate between. To assure that the keyframing algorithm has data at all times, the time delay of the follower has to be greater than $2x$ the sampling rate. This assures that values are present for sampling. In the time before the delayed behavior starts, the result of the follower behavior is that of a provided default value.

9.4 Boids

A classical method for including dynamics in VEs is the use of Boids. Boids are a form of autonomous characters, where the underlying AI is very simple. Craig Reynolds originated Boids as a way of simulating the flocking of birds [Rey87]. His method sets the stage for the use of *steering behaviors* for many different "simple entities," including human characters [Mil06, Rey87, Rey00]. This method works by the combination of simple steering behaviors together to form complex behaviors. This basic method is used for entities from the "simple" flocking of birds to the complex maneuvers of AI combatants.

The implementation of this classical simple entity method is an interesting example of FRVR for a number of reasons. Firstly, it demonstrates how higher functional objects can be created using FRVR. Using Boids with a flocking kind of application demonstrates interactivity between simulated objects. Each Boid is singularly simulated, but with interaction based on other Boids and potentially the environment. The steering behaviors approach is interesting, as it is a well established technique in the field and is based on the same approach as FRP, the integration of continuous functions.

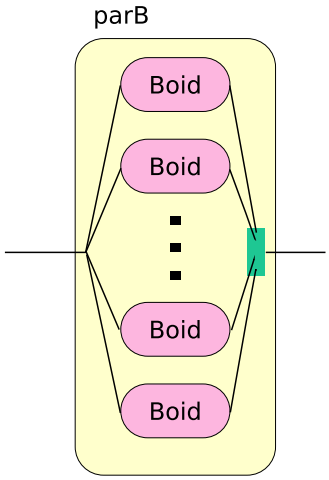


Figure 9.7: The structure of the flock of Boids. Each Boid is an individual SF simulation. The individual Boid outputs are collected into a single output value, a list here.

The final aspect of interest in the Boids implementation is that the functionality is based on a collection of underlying functionalities. This illustrates how complex functionalities can be built up on a component basis using FRP.

Before presenting to the FRVR implementation of Boids, an explanation of the Steering Behavior implementation method is presented. The steering behavior approach builds a seemingly complex dynamic behavior by combining a set of relatively simple rules together. For instance, a bird flying in a flock would have three basic behaviors: separation, alignment, and cohesion [Rey87]. Each of these functions produces a sort of steering information. This is in the form of a desired acceleration based on inputs of the bird's position/orientation and the positions/orientations of the neighboring birds. The separation behavior produces acceleration away from any close neighbors, such that the birds do not collide. The closer they are the larger the "repelling" force. Cohesion is the force that makes the bird desire to be in the center of the flock; it is what holds the flock together. The alignment behavior steers the bird to fly in the same direction as its neighbors. Combining the produced "desire accelerations" via some weighting factors produces an acceleration for the bird at every simulation step. Those accelerations are then integrated to get the velocity and position/orientation.

The basic implementation of Boids in FRVR follows closely the standard implementation of Boids. The implementation presented here is based largely on Millington's more general "steering behaviors" implementation described in [Mil06]. Figure 9.7 shows this design graphically. First, we will assume that each Boid entity is a behavior that controls its internal values, e.g. its position. Each Boid is in its own right an autonomous entity. Therefore, only a single Boid implementation is required. The flock is, as in reality, just a group of Boids. The "flocking" behavior is achieved by each Boid reacting to the rest of the flock.

The Boids are individually simulated entities, though they require information over the entire flock. Although this will be verified below, we can assume that every Boid requires the exact same input. Therefore, the *parB* (**par**allel **B**roadcast) parallel composition wrapping behavior can be used. The *parB* function takes a single input, a collection of SFs (a data type of the Haskell ***Functor*** type class, e.g. *list*), and generates a single parallel SF. The single input is "broadcasted" to each of the behaviors in the collection. A simple *list* for the collection type is sufficient and easiest for the basic flocking behavior. Each Boid calculates its reaction to the flock (part of the input) and generates its new position and orientation each frame. The *parB* SF returns a *list* of results.

```
flock :: SF () [Coord3ff]
flock = proc _ -> do
    let calcGroupCenter :: [Coord3ff] -> Point3f
        calcGroupCenter flockCoords = translatePoint originPoint (vec2Center
flockCoords)
        -- for each boid find vector from self to center and average
        vec2Center boid_list =
            (sum (map ((differenceVector (originPoint)).coordPosition)
boid_list))
            ^/ (fromIntegral (length boid_list))

    rec
      -- simulate the flock
                -- all get the same input, flock positions and flock center
       boid_list <- parB (makeBoids 10) -<
                    (boid_list,
                      (calcGroupCenter (map boidCoord boid_list)))
    returnA -<  map boidCoord boid_list
```

Listing 9.13: Simulation of a Flock of Boids

The developed flock of birds exampled uses the following steering behaviors:

wander add a random component
separation avoid collision with flock mates
cohesion move to the middle of the flock
velocity matching match the flock velocity
alignment match the flock direction

This behavior set contains most of the basic steering behavior components. The implementation of each is relatively straight forward, but it is informative to look at the input requirements of these components. Wander requires only a random input. This is achieved by providing a seed at creation time to a stateful process in Haskell. Therefore, it requires no input each frame. Separation requires the distance of the current Boid from all of its neighbors. Cohesion requires the distance from the Boid to the center of the flock. Velocity matching is developed here based on the average velocity of the whole flock. Similarly, alignment is derived based on the average direction of the whole flock.

The inputs for separation, cohesion, velocity matching, and alignment require comparison of the processing Boid with the rest of the flock or a factor derived from them.

In this area, the Haskell/Yampa solution requires a different solution than the standard solution involving pointers and global variables. Since we have defined the flock as a *list* in Haskell, we can exploit Haskell's strengths to succinctly implement the actual calculations required. Finding the distance between each Boid is a simple calculation, but has to be done many times. The simplest implementation of this means providing the complete list as an input to the separation behavior. Then the ***map*** function can be used to run the same calculation over every entity of the list. A collection of forces derived from the inverse distance is calculated and summed. Similarly, cohesion, velocity matching, and alignment all require flock based variables. In these cases, they are just simple averages but are best performed on the immediate neighboring Boids. To this end, a sphere of influence is defined. Again Haskell's ***map*** makes easy work out of this, given the complete Boid list. First, the list is filtered for the nearest Boids and then the desired accelerations are calculated based on the position/orientation of those Boids.

```
-- try to get to the group center
cohesive <- group_cohesion <<<
            iPre (iBoid, replicate 10
                           (coordPosition (boidCoord iBoid)))
                 -< (boid,
                     map (coordPosition.boidCoord) boid_list)

-- don't get to close to neighbors (no collisions!)
separate <- separate_group <<<
            iPre (iBoid, replicate 10
                           (coordPosition (boidCoord iBoid)))
                 -< (boid,
                     map (coordPosition.boidCoord) boid_list)

-- add a random nature to it all
wandering <- wander 1 <<< iPre iBoid -< boid

-- find the average velocity of neighbors within 2 meter
neighbor_vel <- arr3 calc_neighbor_vel -< (2.0, boid, boid_list)

-- orient to the nearest neighbors (y is up)
desired_rot <- align2Vector (Vector3 0 1 0) <<<
               iPre (iBoid,  vector3 0 0 (-1))
                    -< (boid, neighbor_vel)

-- match the velocity of the nearest neighbors
desired_velocity <- velocityMatching 0.5 <<<
                    iPre (iBoid, (boidVelocity iBoid))
                   -< (boid, neighbor_vel)

-- have the boid face the direction it is traveling
face_dir_travel <- face3D (Quat 0 0 0 1) <<<
                   iPre (iBoid, boidVelocity iBoid)
                  -<  (boid, velocity)
```

Listing 9.14: Steering Behavior Calculations

Each of the basic steering functions provides an acceleration value. Following Millington's suggestion, these are internally scaled to the Boid's maximal acceleration. The next step is to combine the values in a meaningful manner. Various methods can be used for this step. The most flexible and robust method provides a weight for each

value and sums them. The value is clamped to the maximal velocity. A simple even weighting is performed in our example, each contributing a fifth to the acceleration. The code and helper function for the calculating the complete steering input are:

```
    desired_steering <- arr accum_steering -<
                  (weightSteering separate 0.2) :
                  (weightSteering cohesive 0.2) :
                  (weightSteering desired_velocity 0.2) :
                  (weightSteering desired_rot 0.2) :
                  face_dir_travel :
                  (weightSteering wandering 0.2) : []

  where accum_steering steering_list = steeringInput {
           steeringAccelDirection = (accum_accel steering_list),
           steeringOrientation = (accum_rot steering_list) }
        accum_accel steering_list = foldl cap_accel_value zeroVector
                                    (map steeringAccelDirection steering_list)

        accum_rot steering_list = QuatRot (product
                     (map (toQuat.steeringOrientation) steering_list))

        -- cap rotational velocity to (-2pi, 2pi)
        cap_rot_vel w | ((abs w) >= 3.14) = (signum w) * 3.14
                      | otherwise         = w

        -- cap acceleration to max
        cap_accel_value current new |
                   (dot current current) < (maxAcceleration iBoid) =
                                        current + new
                                 | otherwise = current
```

Listing 9.15: Combing the Individual Steering Behaviors

Given the final desired rotational and spatial accelerations, the respective velocities are found; then, new a position and orientation is derived. This is performed with the same methods as in Sections 9.1 and 9.2.

Finally, the input and output of the Boid simulation has to be addressed. Based on the discussion of the requirements for the individual steering behaviors above, the input for each Boid is specified to be *([Boid], Point3f)*. The output of the Boid is simply its essential data. The *Coord3* data (position and orientation combined) is sufficient for display of the Boid, but we have seen that other information is required for the flocking behaviors. Specifically the vector that describes the Boids direction of flight is required. Because of the interactions between the individual Boids, the complete essence of the Boid has to be provided as an output. The Boid's data is fed back into the input of the other Boids in the next frame. For this reason, the data definition of a Boid is returned. The definition of the Boid data type is seen in Listing 9.16.

```
data Boid = Boid {
 boidCoord :: Coord3ff,       -- ^ The Coordinate of the boid
 boidVelocity :: Vector3f, -- ^ The Velocity of the boid
        -- required for some calculations, but could be dynamic
 maxAcceleration :: Float,  -- ^ max accel of the boid
 maxRotationAccel :: Float -- ^ max rotational accel
}
```

Listing 9.16: The data of the Boid data-type. This information defines the Boid.

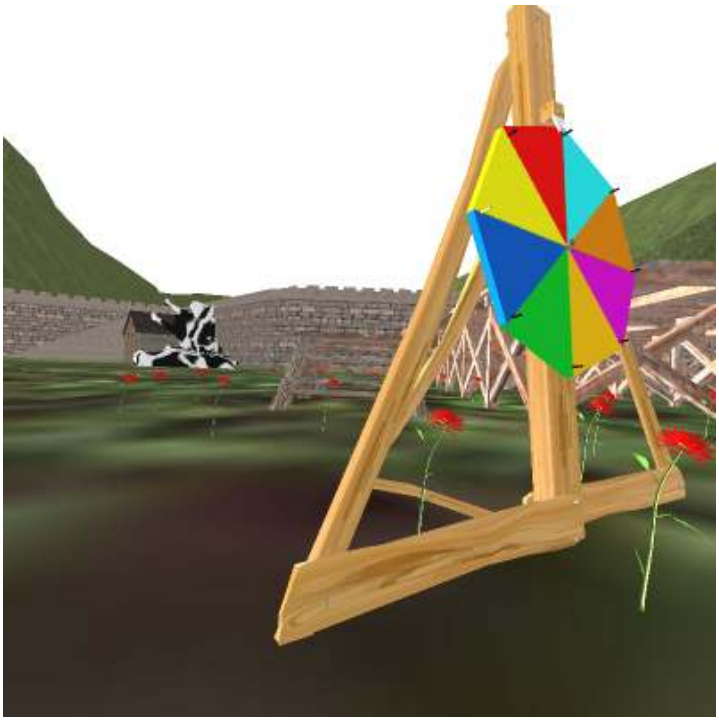


Figure 9.8: Middle Age Castle Environment Basis for an Interactive Experience Driven by FRVR

9.5 A Full DIVE

This sections presents the development of a complete DIVE. The environment is composed of the different examples previously described and many more behaviors and functionalities. In many senses, it is like the Crayoland VE described in Section 2.1.2. It is designed to be an interesting world to be explored. The environment is based on a middle ages castle, developed as part of another work [BB06]. Various objects have been made dynamic and interactive using the FRVR system. Figure 9.8 shows the initial and main area of interaction in the castle. The example shows that, now, FRVR can be used to simulate an environment of multiple DIVE components in real-time and an environment that has multiple levels of behaviors involved.

In the center of the castle is a market place. A number of the small examples that were developed previously are found. On one of the stands is the Newton's Cradle and the simple pendulum. Also found there is a spinning wheel for a luck game. The user can interact with the wheel; depending where the spun wheel ends up, changes to the world occur. For instance, if the blue section of the wheel ends on the spot, a cow is launched from a tower overlooking the castle into the marketplace area. The dynamics of the cow are controlled by FRVR and have a random element. The red color makes flowers grow in the open field near the market place.

The interaction with the wheel is also controlled by FRVR. Here, a not yet addressed functionality of Yampa is used, the derivative. The user spins the wheel, just like in reality. When selected, the wheel can be spun using a direct interaction method; in this case, movement of the wand cause movement in the wheel on its axis. When the wheel is released, the derivative of the rotational movement caused by the interaction causes an impulse rotational velocity on the wheel in a new dynamic behavior. That rotational velocity and a frictional value lead the wheel to spin. After the spinning, new behaviors are started based on the final resting angle of the wheel.

The birds of the previous section fly around the castle. Following the path that leads out of the castle to the river crossing, the windmill can be seen turning in the wind. This uses a simple behavior created by keyframing a full rotation and looping it.

Exploring inside the castle walls can lead to more surprises. The castle turns out to be haunted. The explorer of the castle will soon have a ghost following them around, everywhere. The ghost is developed using a the combination of a few different techniques. Once the user enters the ghost's vicinity it is triggered to start following the user. This is done with the time delayed behavior developed in Section 9.3. This method controls the position of the ghost as it follows the user around. However, this does not address making the ghost move internally. This is achieved via the incorporation of an traditional animation system. Using a traditional animation system makes sense in cases of mesh transformation, as many values have to be changed, and the systems already can handle the complex task of loading and manipulating the control points of the mesh. What the behavior does though, is control which of the animations is running and at what speed.

Using the mechanisms provided by an animation system, in this case the Cal3D animation system [cal], the speed of playback can be manipulated and different animations can be chosen from. The speed of playback can be controlled by the FRVR behaviors by defining a playback factor that is related to the speed of the ghost at the moment. The ghost used, has three different animations: a moving animation and two "bored" animations for when it is idle. The ghost behavior programmed not only follows the user, but in moments of inactivity becomes bored. This is expressed through the playing of other animations.

Getting rid of the ghost following around is only possible by turning back the hands of time. Luckily, such a functionality is possible in a DIVE and easy to create with FRVR. In the tavern, a grandfather clock can be found, ticking away the time. The pendulum of the clock ticks away the seconds and the hands move. The user can, however, turn back time, simply by turning the clock hands backwards. The complete simulation environment is contained in a undoSF that is triggered to save every minute. When turning back the minute hand of the clock the user causes an undo every minute.

9.6 Discussion

The examples demonstrate a diverse variety of Dynamics, Dynamic Interactions, and Interactive Dynamics. The flexibility and power of the FRVR system are shown the examples. The ease of programming physical based systems is demonstrated already with the initial, trivial moving box example. That example also explains how a complete FRVR application is developed. Although FRVR involves two quite different systems, the tools provided make the design, programming, and compilation of applications easy to perform. The problem is divided into steps that are well supported. More complex systems, like the Newton's Cradle, are easy to build up from basic components, i.e. pendulums in the case of the Newton's Cradle. The fit of the FRP approach to the nature of DIVEs is shown throughout the examples.

Returning to the use cases collected for FRVR in Section 7.1.1, we can see that they are represented in the examples provided. Examples of each of the use cases can be found in the examples as follows:

Path Recording/Playback the follower of Section 9.3.

Creation/Deletion of Objects in a Scene cows and roses of the castle in Section 9.5.

Spatial Behaviors launched objects of Section 9.2.

Inner-object Changes balls of the Newton's Cradle in Section 9.2.4.

Entities Boids style Simple Entities in Section 9.4.

Story the castle environment of Section 9.5.

Time turning time back (undo) in the castle environment of Section 9.5.

Direct Manipulation Pendulum of Section 9.2.2.

Higher-order Direct Manipulation spinning wheel of the castle environment of Section 9.5.

In addition, a number of Interactive Dynamics are present: Newton's Cradle, follower ghost of the castle, and time in the castle. These examples demonstrate the ease in which complex behaviors of Interactive Dynamics can be created. The follower ghost of the castle is worth extra note. It is an indirect interactivity, though the user's movement. It is also a Dynamic Interaction with a Dynamics.

Finally, the ability to build up complex worlds was shown by the castle environment. All of the components of it were developed individually. This means each can be reused in other contexts. The ability to build DIVEs component wise is also necessary for the future development of massive environments that many of dreamed of creating. Although the castle is limited in scope, it demonstrates how large environments can be created and how story dependent changes to the world are created.

Chapter 10

Conclusion

The creation of Virtual Environments that go beyond the classical static worlds that populate much of Virtual Reality has been the focus of the research presented. Two ways of reaching beyond static worlds to engage users were identified and pursued: the introduction of dynamics and the introduction of interactions. The design space of those components and their combination has been explored and analyzed. Informed by that work, a system of support has been developed, the Functional Reactive Virtual Reality (FRVR) system.

This chapter summarizes the results of this dissertation and provides further directions work in this field could take. The first section discusses the results of this dissertation and their impacts. Section 10.2 lists the contributions of this work to the research community. Section 10.3 presents some of the avenues for further work that have been identified.

10.1 Discussion

Years of working with engineers and other scientist and watching them struggle with the adaptation of their work to the systems of VR led to the impulse to start this work. Even those with extensive programming experience struggled to understand the concepts that underlie the VR way of application development. After a few days, they would be able to correctly simulate something very simple, like an object falling. However, the simulated environments that they had envisioned creating, were inevitably abandoned for very simple worlds, with very limited capabilities. Even the author, a Computer Scientist, struggled with the methods present throughout various systems and wondered at the sterile environments that seemed to be the standard throughout the VR community.

This work has addressed those needs with a new approach to creating such environments. It has also reached beyond the needs of the original motivation. The FRVR system, developed in Part II, is based on mechanisms for the creation of time depen-

dent behaviors that are defined in a traditional mathematical way. FRVR bases on a programming paradigm that directly supports the needs of creating VEs that are composed both of continuously changing components and discrete and continuous interactions, i.e. a hybrid system. In FRVR, behaviors of objects in the VE are programmed as continuous functions and a discrete reactive nature enables interactions and the building of complex system. This hybrid approach matches the developers understanding of the behaviors they are programming. The effectiveness, power, and flexibility of FRVR have been shown through examples that range from simple dynamics and interactions to a complex environment composed of diverse behaviors.

Although this work is not unique in its focus on the area of supporting the creation of environments that are interesting, interactive experiences, it is unique in its approach to building this support. This work is founded in an exploration and analysis of the type of environment of interest. It is the first work in this area to define and investigate the nature of the environments that are to be supported. We refer to those environments as Dynamic, Interactive Virtual Environments (DIVEs). It is also the only work that has surveyed the potential approaches that could be taken in supporting the implementation of time dependent phenomena. A survey of the approaches already taken by others and discussion of their effectiveness completes the background research of this work. Together, they form a solid foundation for understanding the area, the difficulties that lie therein, and how support can better be approached.

The next section lists the major scientific contributions of this work. One of the more lasting impacts of this work may come from the expanded understanding of the area exposed in this work. The breadth of the various areas that informed this work and connections made between them have led to the development of an approach that is uniquely suited to the problem space, but also to the identification of many potential areas of future work. The area of DIVEs is currently experiencing a renaissance in the VR community, due in part to this work. Coupled with the emergence of gaming consoles that exemplify the potential of this direction, the outlook for research in all facets of this interesting area is good. This work has laid foundations for future research in all of those facets.

10.2 Contributions

This section highlights the contributions of the dissertation to the research community. The contributions are presented in two sections, stemming from the investigation of DIVEs and the development of the FRVR support system. The contributions are listed roughly in order of their significance within the respective sections.

10.2.1 The Dynamic, Interactive Virtual Environments Design Space

The investigation of the design space of environments that can be considered compelling, experiential Virtual Environments contributes to the community in a number of ways.

Dynamic, Interactive Virtual Environments

The investigation and formalization of the kinds of environments that are interesting and exciting for the user is an important step and valuable contribution to the community. We have defined such environments as "Dynamic, Interactive Virtual Environments," a name that evinces the components of importance: time dependent dynamics and interaction. DIVEs were defined in Chapter 2. A major contribution of this work is the formalization of the design space of these environments. This provides both a framework for exploring the manifold potentials of DIVEs and a foundation for the establishment of requirements for a system to support the creation of such environments.

Dynamics Design Space

The exploration and categorization of the design space of potential Dynamics in VEs exposes the multitude of such phenomena and also the nature of such possibilities with respect to their defining characteristic, time. The developed taxonomies inform the design of the FRVR system and also can be used in the VE authoring process as a guideline to the possible Dynamics. The Dynamics design space was explored in Chapter 4.

Interactive Dynamics, Dynamic Interaction, and Dynamic Interaction with Dynamics Design Spaces

The formalization of the DIVE design space exposed three content areas that have not yet be considered in the literature: *Interactive Dynamics*, *Dynamic Interaction*, and *Dynamic Interaction with Dynamics*. These content areas sit at the conjunction of Dynamics and Interaction and are first identified and explored in this work. These areas of are importance, because of the significant effect we expect they will have on the experience of DIVEs. They are also completely open areas of research; This work was only begun to touch on the potentials they present. Chapter 5 defined and explored the design spaces of these areas.

Time and its Representation in Computer Sciences

The review of time and its representation provide a basis for the work. A survey of the handling time dependent phenomena in Computer Science areas forms a contribution of this work. A review of ways to deal with time helps provide an understanding of the problem of the specification and implementation of time based phenomena. It provides a basis for selecting methods for support of such systems. The review of time and implementation methods survey can be found in Chapter 3.

10.2.2 Software Framework to Support Creation of DIVEs

The creation of a software framework to support DIVE creation has also provided a number of contributions.

Functional Reactive Virtual Reality

A major contribution of this work is the Functional Reactive Virtual Reality (FRVR) framework. It is the only system to date to be explicitly designed to support DIVE creation. FRVR is also the only system to be built on requirements derived from an analysis of the domain space. FRVR is built on a programming paradigm that is well suited to the hybrid nature of the DIVE space, Functional Reactive Programming. FRVR uses a modular design and is cross platform and cross VR software. The design used in FRVR was presented in Chapter 7 and published in [BB07a]. Other publications covering this work include the papers [BB07c, BB08], which explore how the FRVR system, using FRP, enables the creation of DIVEs, and [BB07b], which explores FRVR's cross platform capabilities, as well as showing how FRVR works complementary to the high-level capabilities already present in some VR systems. FRVR has been released as open source via the popular SourceForge.net web portal[1].

FRVR Extensions to Yampa

A series of advanced and DIVE specific functionalities that extend the usefulness of the Yampa FRP framework have been created in Chapter 8. They include classically desired functionalities, like time skewing. The most important and interesting of the extensions is the *undo* capability. The undo implementation exploits Yampa's unique design to create a true undo for any type of simulation component including the ability to undo continuous interactions. This is significant, as no other system supports such an undo mechanism.

Visual AFRP System

A Visual Programming Environment was developed, in order to simplify the creation of Arrow based Haskell code. This work was performed as part of a diploma work under the author's guidance and is presented in Appendix B. One of the motivating factors in the development of this dissertation was creating a system of support for DIVE creation that was easier to use, particularly for non computer scientists. While the Yampa system brings many advantages with its use, the syntax of its Arrow based code can be difficult to adjust to. The Visual Programming Environment lightens the difficulties in learning and developing Arrow based code (that Yampa depends heavily on). The developed VPE is capable of producing Yampa code or pure Arrow code, so that this contribution can also be used by a wider community.

[1] http://sourceforge.net/projects/frvr/

Survey of Existing High-level VR System Support

This works presents a survey of relevant systems that support the creation of combined dynamics and interaction content. The survey covers the breadth of the different approaches and focuses on the most relevant systems. This survey is the only existing survey to focus on the higher-level abilities of the systems that allow them to create DIVE like environments. Existing surveys of significance focus largely on software engineering approaches taken in the systems reviewed. The survey can be found in Section 6.2.

Practitioner Oriented Explanation of Functional Reactive Programming

Introductions to the Functional Reactive Programming paradigm, the Yampa system, and the Haskell language are included in the dissertation in Appendix A. Because the resources available are written for a specific community, the (functional) programming languages research community, they are not very accessible for the general public. Therefore, the introduction provided is expanded to form a basis for understanding these complex paradigms and systems from the perspective of a user of the system. This contribution is a boon to parties interested in using FRVR, FRP, or Arrow based Haskell systems. It also helps prepare the reader for further investigations into the existing materials.

10.3 Future Work and Research Directions

A number of research directions and further work potentials have been identified during the course of the dissertation. Here, a selection of the possibilities are presented. The first sub-section suggests research into the validity of the conjecture that DIVEs truly are better than static environments. The second sub-section describes interesting research directions into the newly identified areas of Dynamic Interaction and Interactive Dynamics. The third sub-section discusses further work that could be performed to improve the developed FRVR system.

10.3.1 Validity of the DIVE Conjecture

A fundamental conjecture of this dissertation is that DIVEs provide a more engaging user experience than a static environment. This conjecture is an implicit underpinning to the motivation of this work. This conjecture was assumed valid for the purpose of this dissertation. Across the spectrum of related real-time VE areas, there is a widely accepted, yet implicit, assumption as to the validity of this conjecture. Some of the most respected VR researchers [SVS05, Whi03] have asserted this is true. Unfortunately, the fact that this is just an assumption is rarely recognized. The conjecture that DIVEs are better than static, non-interactive VEs needs to be formally validated.

Formal testing of the hypothesis is difficult, due to the various factors that have to be considered, and, therefore, fell outside the scope of this work. However, we have already started research in this direction. We have started to consider the question of how DIVEs actually affect the user. An initial hypothesis was that DIVEs increase the user's presence in VE over static environments. While this seems a reasonable result to expect, there are a few concerns that surface when considering the topic in depth. Ideas from this theoretical exercise were presented as part of the "Peach Summer School" on presence [Blo07a, Blo07b].

Those publications also proposed that Norman's idea of object affordance [Nor02] is also valid for virtual objects and that unmatched affordances effect presence. Moreover, in [Blo07b] we proposed that dynamics in a VE afford interaction. It seems there is something in human nature that makes us want to "play" with anything that is moving, at least in a virtual environment. This theory is largely based on personal observation of the authors over years of monthly demos, where any moving object led to the guests asking if they could interact with it, regardless of the physical feasibility of such an action. If this theory is correct, the impact of dynamics would be large and manifold. Beyond the theoretical exercise presented at the Peach Summer School, we have begun research into these directions. Current research in our group is attempting to ascertain the applicability of Norman's affordances idea to virtual objects.

10.3.2 Research on the Confluence of Dynamics and Interaction

One of the largest results of this dissertation is the identification of two major research directions that have had little attention, Dynamic Interaction and Interactive Dynamics. Both of these areas are wide open for further research.

The term **Dynamic Interaction** has been introduced in this work and the design space identified. Many of the possible interactions that fit into this category are well established concepts within VR and related fields. Almost all of the interactions considered in Bowman's Interaction Taxonomy [BH99], the pseudo standard in VR interaction research, can be found in our analysis of the design space of Dynamic Interactions in Section 5.1. However, only Steed has noted this time dependent nature of interactions, and then only in connection with selection [Ste06]. The foremost direction that needs to be addressed is an analysis of interaction techniques that explicitly considers the time aspect. In this work, the time aspect of dynamics interactions was considered, but only in terms of developer perception and the their implementation. Another research direction is the development of new interactions with the time component explicitly considered. We believe this should result in better interactions.

Perhaps the most interesting and mostly widely open direction for future research to be identified in this work is the area of **Interactive Dynamics**. This large topic has been only partially addressed in this work; The research potentials in this direction are vast. The issue of interaction with dynamics has largely been ignored by the VR research community, and the computer game community has not yet moved to more formal research of this area. We believe that one of the core reasons for this is that

Interactive Dynamics is a difficult issue, at the conceptual level and the implementation level. Moreover, the physical counterparts to Interactive Dynamics in the "real" world are not particularly easy. For instance, many sports (notably "ball sports") center on interaction with dynamics. This typically involves a dynamic object (the ball) and also the players. Given the difficulty of such interactions in the real world, Interactive Dynamics in the virtual environments can be expected to be difficult.

Because it is an area of such potential, work on Interactive Dynamics is already in development in our research group. As most interactions with dynamics are dependent on selection (indication of the object of interaction), selection is the most logical starting point for the exploration of this space. Initial results from a study performed as part of a German Diploma thesis by Roland Schröder-Kroll can be reported on and will soon be published [SKBB08]. That work explores the selection of dynamic objects. Three new selection techniques were designed specifically to address the needs of selection of dynamic objects. These techniques attempted to help the user by constraining the selection parameters. One technique constrained the selection, by implementing a snap-to pointer. Therefore, the nearest selectable object was pre-selected. Another technique used a cone selection method, where part of the criteria for the selection was the length of time the object was in the cone. The final technique constrained the selection to be the object moving at the same velocity that the user moved the wand. A pilot study has been performed, evaluating the usability of those three techniques in comparison to the standard ray-picking technique. The initial results suggest the cone selection and snap-to methods to work best under conditions of selection of single dynamic objects, alone and in a field of similarly moving obstructing objects. More information can be found in [SKBB08].

10.3.3 Improvements to FRVR

The developed FRVR system can be improved on a few fronts. The provisional data exchange system and the lock-step design are two changes proposed here. The data exchange system was written as a proof of concept, replacing it with a better developed solution would improve the complete system. Changing from the lock-step process is more experimental, but has a number of potential benefits.

Data Exchange System

There are three possible directions in replacing the data exchange system that are of interest. These directions were either directly or indirectly considered in the design and development of FRVR and the data exchange in Section 7.3.2. First, the development of a more robust and complete data exchange based on the shared memory would benefit FRVR's long term usability. This option should be adopted if the other two are not. The other possibilities involve moving to a networked solution.

The movement to a **networked** based system would be justified by the advancements discussed in the next section. In short, this would allow a multi-platform and distributed system to easily be built. Networked data exchanges are well understood,

though with many implementation challenges like synchronization and lag. A slightly older, VR specific reference work to networked simulation is [SZ99]. An interesting alternative is to build a distributed BlackBoard system, like the ones that inspired the data exchange system developed. A good overview of the area can be found in [Cor91]. When moving to a networked data exchange system, the possibility of expanding to an **agent based BlackBoard** should be considered [Cor03]. The idea of the agent based systems is that the individual systems work as agents in the complete system. Agents request, via the BlackBoard, that another agent handle tasks they can not handle. For example, the simulation could request that sound "squaw" be played. The audio agent would commit to this and perform the task. This has some interesting possibilities for advanced situations, particularly for dynamic load balancing and dynamic heterogeneous system deployment.

Free-running Threaded Architecture

The basic architecture of the FRVR system was selected from three different architectural designs for the integration of FRP and the VR systems in Section 7.2.2. The FRVR implementation builds on a lock-step architecture. For four main reasons, it might be advantageous to move to a free-running threaded architecture: scalability, simulation resolution, multi-output systems, and input system processing.

The free-running approach would allow the creation of a multi-machine distributed system. Such a system would be very **scalable**. The simulation "space" could be divided and distributed across multiple machines with only minimal more effort than the current single machine system. As most dynamics are independent of each other, such a solution should be effective. Moving to a free-running multi-threaded architecture allows the FRP simulation to be run at any **simulation rate**. The simulation can be set to run at whatever update rate is required to achieve simulation stability using Yampa's original simulation kernel. Where the independent simulation resolution becomes very advantageous is in multi-modal output systems and for direct input systems.

Multi-modal systems are of ever increasing importance in VR and related fields. The challenge of the multi-modal systems, from a software side, is that each modality has different real-time requirements (see Section 6.1 for more discussion of this). Because all of the sub-systems have to run at different speeds, such systems are almost exclusively written as decoupled, multi-threaded systems. Decoupling the simulation from the output modality means that each modality can run at their appropriate update rate, receiving new simulation updates as they are available.

If the environment is defined in the simulation side of FRVR, then an interesting future change to the system is incorporating the **input system** directly into FRVR. Moving to a free-running simulation with integrated input from the interaction devices would bring several potential advantages. The foremost is that the simulation would be able to run on all the input data available and with minimized lag. Dynamic Interactions in particular are highly susceptible to these parameters. This addresses classical problems in VR input systems, e.g. lag and missing data.

Part III

Appendix and Bibliography

Appendix A

Functional Reactive Programming

The technological basis for the FRVR framework of support developed as a core of this dissertation is the Functional Reactive Programming(FRP) paradigm. The design phase identified FRP as the system to base this work on, as it provided many of the functionalities required in a unique way (see Section 7.2). The concepts that make FRP well fit to the problem of supporting DIVEs were introduced. This Appendix provides a more thorough introduction of the FRP paradigm and the important implementation details for this dissertation. This presentation of FRP is unique in that it is the only presentation of the paradigm and foundations that is written from a non programming language researcher vantage point.

The FRP paradigm will be further described in the first section and the context in which the paradigm was developed is introduced in section that follows (Section A.2). As the implementation is highly dependent on the Haskell language and its constructs, an introduction to Haskell and the Arrow concepts is provided in Section A.3. Section A.4 provides an explanation of the Yampa FRP system and the functionalities it provides. Finally, a few of the application of FRP to related problems is discussed in Section A.5.

A.1 Concept

Functional Reactive Programming (FRP) is a programming paradigm that is characterized by its two-fold nature; it is composed of continuous behaviors (functional) and discrete events (reactive). At its core FRP is concerned with the modelling and calculation of time dependent systems. FRP is, therefore, a paradigm designed for Hybrid Systems [PH03] (see Chapter 3 for information to Hybrid Systems and related areas of reactive systems). The FRP's continuous functionality is defined and programmed via integral mathematics. The reactionary functionality of FRP is formed through

the ability to replace the running continuous functionality with new functionality on a discrete event occurrence.

FRP is implemented as a Domain Specific Language(DSL), usually in the functional programming language Haskell. What this means in this case, is that specially developed programming support is embedded in to the language to specifically support the hybrid model of FRP. In terms of usage, FRP is essentially a special simulation environment embedded into a standard Haskell environment. FRP is, in essence, a completely closed environment that is run by a special kernel. In looking at the implementation, FRP support as a simulation model that is similar to that of continuous simulation community (see Section 3.4). One difference is that FRP has always been designed with an explicit goal of providing a soft real-time simulation. This can be seen in the application spaces FRP has used as examples; these applications are highlighted in Section A.5.

However, within the developing community FRP is not viewed as a Hybrid System solution (excepting by Pembeci et al. who identified FRP as such [PH03]). The functional portion of the name is often identified with the functional programming paradigm, in which most implementations are created. Excepting one [DHP02] (written in C++), all implementations have been programmed in the Haskell language. More formally, the community view can be seen in the definition of FRP properties Antony Courtney, creator of the current FRP version, provided [Cou04]:

1. purely functional,
2. first-class time-varying values, and
3. declarative reactivity.

The purely functional property says that the output at any time relies only on the input at that time. This is reflection of Haskell, a pure functional language. We will see, however, that this does not necessarily hold true in the current FRP system, Yampa. First-class time-varying values refer to two aspects. Time-varying values indicate continuous valued variables that are time dependent. The values are first class values, meaning they can be used the same as any other data, e.g. stored in variables or passed as arguments. The declarative reactivity is that they system reacts to events in a manner like declarative programming would specify.

A.2 Paradigm Context

The FRP paradigm was introduced by Conal Elliot, who was motivated by the desire for a method of enabling the modelling of computer animation that more closely matched the human perception of the motions that were to be animated. Computer animation, at the time, was performed largely by describing the animation in terms of discrete steps and interpolating between them, i.e. keyframing. Elliot felt that animation should be modelled continuously and the presentation of the animation should

handle the discretization of the continuous signal. Elliot also included interaction in his development in the form of sampled continuous interaction and discrete generated events; more precisely he includes mouse movement and buttons presses respectively.

Elliot's initial implementation was the TBAG system, presented in 1994 [ESYAE94]. TBAG was a constraint based system, written in C++ and introduced the initial ideas that lead to the FRP paradigm. *Behaviors*, Elliot's continuously modelled functions, were data types, representing functions of time. The composing of an animation and in a larger scope, the scene, was implemented using a constraint network that was driven by the SkyBlue constraint solver (constraint networks are described in Section 6.2.2). The current values were generated by the user inquiring the system for the values at time t. The constraint system calculated the values based on the network written. Discrete interactions are actually only handled at a higher level, in what Elliot termed Manipulators. The Manipulators receive behavior-generated values, constraining them further based on the discrete input and are external to the Behavior system. The TBAG implementation is a predecessor of and very similar to several VR systems [BG95, Del00, JDM99, TLG99, WGW90] described in Section 6.2.2.

The Functional Reactive ANimation(FRAN) system was the next generation system created by Elliot in collaboration with Hudak [EH97]. FRAN took a major step in changing the implementation language from C++ to Haskell and moving away from the constraint based system. Haskell is a lazy evaluation, strongly typed, pure functional language [Has98]. A discussion of relevant portions of the Haskell language is presented in Section A.3, where further references are provided. As in TBAG, the behaviors are data types representing pure, continuous-time functions. FRAN introduces the discrete *event* concept into the implementation, making it the first true FRP system. Events are discrete occurrences that may have a value attached to them.

FRP has further evolved from FRAN, through a number of generations and influences, as it has been applied to various area. This evolution is well detailed in [Cou04]. The most important implementation change in our context came with the introduction of AFRP, later renamed Yampa [CNP03]. Yampa incorporated Arrows in order to overcome some of the limitations of previous versions. The most recognized change was the alleviation of "space-time" leaks. These crippling leaks caused space leaks, memory leaks familiar to programmers, as well as losing time in the behavior calculation. The implementation referred to as SOE FRP and even FRAN suffered from "space-time" leaks. These leaks effectively limited the usefulness of the implementations to trivial examples laboriously coded. In attempting to control the leak problem, the SOE FRP implementation restricted the expressiveness of FRAN [Cou04]. Yampa manages to reduce the possibility of leaks and reinstates the expressiveness of the FRP concept. Section A.4 describes the Yampa implementation in greater depth.

A.3 Haskell and Arrow Programming

The Functional Reactive Programming paradigm is implemented in the Haskell programming language. In this section, a short introduction to the Haskell language is provided. The introduction focuses only on those aspects that are vital to understanding the content of this dissertation at a superficial level. Following the Haskell introduction, a second subsection is dedicated to the Arrow type class extensions to the Haskell language. The Arrow concepts are presented in more depth, as they are crucial to the Yampa system and the developments of this dissertation. References to in-depth materials are provided in those sections.

A.3.1 Haskell

The Haskell programming language is a pure functional language. Haskell has traditionally been viewed as an "academic language." However, the language is standardized, and a revised second standard is in preparation. The sophistication and robustness of the current generation of compilers have brought Haskell to age. In this section the more relevant aspects of Haskell to this dissertation are presented at a high level. The reader interested in the details of the Haskell language should refer to one of the various books on the subject, [Bir98, Hud00, Hut07], or the Haskell '98 Report [Has98] for in-depth information.

Haskell shows unique characteristics that differentiate it from other pure function languages. The Haskell community describes their language as:

> "Haskell is a computer programming language. In particular, it is a polymorphicly typed, lazy, purely functional language, quite different from most other programming languages. The language is named for Haskell Brooks Curry, whose work in mathematical logic serves as a foundation for functional languages. Haskell is based on lambda calculus, hence the lambda we use as a logo" `http://www.haskell.org` [has]

Pure functional languages are built on the principle of all processing being performed through functions, where said functions are pure. A pure function may not cause "side effects." That means that the function can only take the input provided as an argument and only affect the value of the output parameter. As such, ideas familiar in declarative languages, such as state variables, are not possible in a pure functional language. A function can be thought of as a box that has inputs and outputs, visualized in Figure A.1. The box contains all the function does and there are no other outputs from the box allowed, like changing a global or even local state. It is important to note that all internal variables cease to exist when outside the function, which means that no state is possible between calls. This pure nature has certain benefits. One benefit is that programs composed solely of functions without side-effects can be formally proven to do what is expected. In Haskell's case this mirrors directly a mathematical proof,

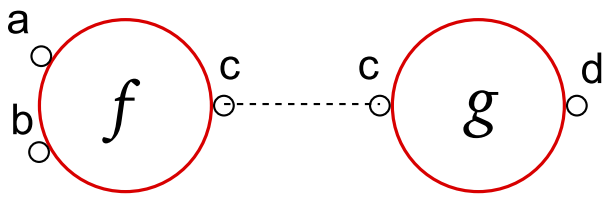


Figure A.1: Haskell functions viewed as filters. However, pure Haskell performs lazy evaluation, precluding a computation view of the boxes, as the execution order is not guaranteed.

often directly and naturally. The second major benefit is that the compiler knows all dependencies and can, therefore, more efficiently optimize the code.

In a strictly pure functional language, there is no way for input into the system, all data must be provided at compile time. There is also no way to get output from a strictly pure functional language. Requiring programs to take neither input nor produce output is excessively restrictive. To counteract this, Haskell incorporates a number of methods for allowing external access. The conceptually simplest of these methods are the input/output (IO) functionalities. IO is a class of functions with which Haskell affects the global state. More simply put, it performs input and output with the outside world. An example of an IO function is the Haskell main function, which retrieves input values, calls any processing functions, outputs any necessary values, and returns a value on ending the program.

The IO functionality is implemented as a *Monad*. Monads are a special class type that comes out of category theory (class types are explained after the Monad discussion). As such, Monads are a bit beyond the clarification required for this dissertation, but a few aspects have to be understood. The interested reader can find numerous tutorials and other explanations at http://www.haskell.org/haskell/Monad. Monads introduce a capability that is essential to a number of functionalities: IO, State, (File) Readers, and (File) Writers. The key to understanding Monads is related to Haskell being a lazy evaluation language, meaning the order in which statements execute is not strictly defined by the code author. The Haskell compiler arranges code so that it is only evaluated if and when it is required. This is a problem for a number of programming concepts, e.g. I/O, state, and file access. IO typically needs to occur in a specific order. State may not be obvious in this, but if we write twice to the variable and read from in once in the middle we want to be able to get the first value, something not guaranteed with lazy evaluation. Monads define basic properties that guarantee the order of execution.

In order to do this a new syntax is introduced, the *do* notation. The *do* keyword is defined as the first part of the function and forces the function to be a Monad type.

The notation specifies a procedural style of programming. The do notation looks like:

```
initDrinkingBirdMover :: IO (StablePtr (PType))
initDrinkingBirdMover = do
                let i = 10
                mutableQuat <- newIORef i
                putStr "before the creation of stablePtr"
                intpointer <- newStablePtr mutableQuat
                putStr "returning teh stableptr"
                return intpointer
```

Monad functionalities allow breaking the purity of Haskell and forcing ordering of calls. As such, they are considered unpure. For that reason any function that calls a Monad function, for instance IO, then takes on the properties of the Monad. That means it is likewise impure. This is done so that the compiler and also the programmer are aware of this. Methods are provided to mark calls as safe. This is an indicator to the system that the call will not cause corruption (e.g. a segmentation fault). Typically this means that the developer has added code to handle any problems gracefully. The usage of these functions is frowned upon, as it potentially makes it difficult to assure the program functions as desired and reduces the compiler's ability to optimize. However, they are included for cases that require them.

The typing system of Haskell is based on type classes. In Haskell all types are data structures. Standard types include the usual programming types, like integers and floating point values. In addition are things like tuples (composition of other types together) and lists. The programmer can create their own types. Programmer declared types are first class types, as their definition is not performed differently than the built in types.

Simple data types are defined as follows:

```
data Boid = Boid {
   boidCoord :: Coord3ff       -- ^ The Coordinate of the boid
   , boidVelocity :: Vector3f  -- ^ The Velocity of the boid
                               -- required for some calculations
   , maxAcceleration :: Float  -- ^ max acceleration of the boid
   , maxRotationAccel :: Float -- ^ max rotational accel of the boid
}
```

Listing A.1: Definition of the Boid Data type, including accessor function names.

Here the data type for a "Boid" is created (see the Boids example of Section 9.4). The Boid has four components of types: Coord3ff, Vector3f, and Float. A unique feature is that the fields of a data structure can be named. That name becomes the accessor function to retrieve that data field. It can also be used to selective set parts of existing data object or in constructing the data object.

Type classes are designed for more complex data types. They are defined by a particular set of functions. Any data type that implements that type class has to have those functions defined to work on its data. For instance, all floating point types have to implement the functions: pi, exp, log, sin, cos, sinh, cosh, asin, acos, atan, asinh, acosh and atanh. Additionally, type classes can inherit from other type classes. The Float data type implements Floating and also the types Data, Enum, Eq, Fractional,

Num, Ord, PrintfArg, Read, Real, RealFloat, RealFrac, Show, Storable, and Typeable. Therefore, it has to also implement all the functions of those type classes. Monads are a type class that define the functions (≫=), (≫), return, and fail.

Haskell is a strictly typed language, so most errors are found at compile time. This however, is not much of a limitation to the language. One of the powers of Haskell is its use of polymorphism. In Haskell, polymorphism means that the type of the function is not expressly given and can work with any data type that has the requisite functions or is of a specified class type.

Haskell's syntax and grammar follow roughly that of mathematics. Although some of the symbology differs, those used to mathematical proofs can pick it up on Haskell's syntax quickly. This is fortuitous for usage by those that are not programming experts, but are knowledgeable in mathematics. In the function listing below one sees a function using a number of these mathematical syntaxes:

```
example_fun :: Floating a => a -> a
example_fun ex_input | ex_input <= 0 = 0
                     | otherwise     =
                         let init_val = 5 * sin 90.0
                         in ex_input *init_val/div_val
                           where div_val = 10 * cos 50
```

Listing A.2: An example function demonstrating the syntax of Haskell. The function is solely to highlight Haskell syntax.

In addition to standard notations like ***let*** and ***where***, this example shows the guard notation used in Haskell. The guard tests the input before other processing occurs and can include many cases. In this case, if the input is less than zero, the result is zero; otherwise the return value is calculated.

Much of Haskell's power and elegance lies in its handling of lists and the usage of higher-order functions. Keystones are functions like map, which executes a run-time provided function on each item of a list. Lazy evaluation is particularly important here, as only the values actual required will be calculated. Therefore, Haskell programs often deal with infinite lists, as only as many values as require will be evaluated. This concept a bit foreign for many, but is part of the power of the lazy evaluation; lazy evaluation calculates only values that are truly needed. In the case of infinite lists (for instance a cyclic list), the list is only accessed for values that are required. Therefore, the list is only theoretically infinite and practically finite. As the map function indicates, functions are often dealt with as values also. These higher-order functions are often passed as parameters to be executed as required in that system. Haskell programming uses higher-order functions often.

The Haskell syntax relies on formatting, both of text and spacing, to determine types, commands, etc. Data types are always written with a capital letter to begin, e.g. Float, Int, Boid. Listing A.2 shows a polymorphic function. The function is polymorphic on the input and output type, which must both be the same. The lower case 'a' value indicates that the value is polymorphic. At compile time, Haskell determines the typing of the calls and insures that the types match. Finally a special syntax, reminiscent of mathematics syntax, allows one to constrain polymorphic types.

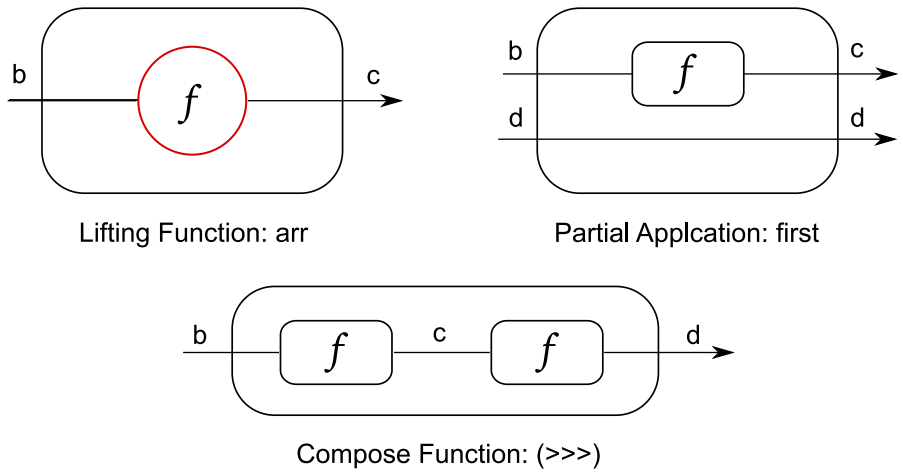


Figure A.2: Diagrams of the three basic Arrow functionalities: the lifting operator *arr* the composition operator ⋙, and the *first* function. The diagrams mirror those found at http://www.haskell.org/arrows.

The listing additionally shows such a restriction, (***Floating*** a =>). The actual call must use data that implements the class type Floating, e.g. Double or Float.

A.3.2 Arrows and Arrow Syntactic Sugar

Arrows are an important extension to the Haskell language. Arrows were introduced by Hughes in [Hug00] and are derived from category theory. Arrows were developed as a way to deal with the external call and call ordering problem present in lazy evaluated pure functional languages like Haskell. Arrows allow the developer to deal with the level of computations, instead of functions. Arrows can be viewed as a generalization of the Monad type class, or rather, the Monad type class is simply a special Arrow instance. An introduction to the Arrow principle and functionalities is presented here. The introduction is somewhat terse, but unique in its external (non-programming language research) vantage point. The reader interested in further details should consult one of the various Arrows resources for a more in depth look [Hug00, Hug04, Pat01, Pat03].

The Arrow type class defines only a few functionalities: the lifting function *arr*, the composition operator ⋙, and the partial application function, *first*. These functionalities are fairly easily understood from their representative diagrams, found in Figure A.2. The composition operator enables specification of data flow through the computations. The partial application operator makes it possible to specify the order of operation of two unrelated operations. This is necessary as, the compiler is still able to reorder non-related operations for speed ups. Combined with the second operator (the mirror image of the first), the order of operations can be forced to be dependent, even when the values are not.

From these basic functionalities, other desirable functionalities can be built. The *second* operator is the complement of the first, where the second argument of the input pair is processed by the Arrow instead of the first. To simplify the processing of Arrows that order dependent, but not directly connected, the *** is introduced. This parallel composition operator is defined in terms of a first and then a second, taking two Arrows and paired input/output. The other operator often seen is the &&&. It is related to the *** operator, but takes a single input value and distributes it as the input to both the Arrows provided. It returns the results paired.

Specializations of the Arrow type class extend its usefulness further. The ArrowChoice type class introduces conditionals. The ArrowLoop type class introduces a recursive nature to Arrows. This recursive nature allows the Arrow to feedback values from the output into the input, allowing an ArrowLoop instance to be stateful. Finally, the ArrowApply class creates the possibility of using higher-order Arrows. These Arrows are capable of receiving other Arrows as input and "applying" the incoming Arrows, i.e. causing its execution within that Arrow's functionality. This is analogous to higher-order functional composition.

Using Arrows forces a point-free style of programming. As noted in various places [Cou04, Pat01], programs written in point-free style rapidly become difficult to read. The listing below demonstrates a function in point-free syntax that will be shown again in Listing A.4.

```
sinf :: Arrow a => a (Float, Float)
                     ((Float, Float), Float)
sinf = ((first (arr sin) >>>
   arr (\ (sinOut, fIn)
          -> (fIn, (fIn, sinOut))))
  >>> (first f >>>
   arr (\ (fOut, (fIn, sinOut))
          -> (fIn, (fOut, sinOut))))
  >>> (first (arr sin <<< f) >>>
   arr (\ (sinOut, (fOut, sinOut))
          -> ((sinOut, fOut), sinfOut))))
```

Listing A.3: An Arrow in Point-free Style.

Paterson introduced a point-wise notational extension for Haskell Arrows to help overcome this limitation [Pat01]. Due to its ease, it is used more often than the point-free syntax. This notation introduces a new Arrow abstraction, *procedures*, and a *do* notation reminiscent of the Monad *do* notation. One of the greatest benefits of the procedure notation for beginning and non-functional programmers is the ability to name values between steps. Not only does it increase the readability of the code, it makes it possible to cleanly and clearly route values through the function. An example of the point-wise style of programming can be seen below.

```
sinf :: Arrow a => a (Float, Float)
                     ((Float, Float), Float)
sinf = proc (sinIn, fIn) -> do
   sinOut <- arr sin -< sinIn
   fOut <- f -< fIn
   sinf-val <- arr sin <<< f -< fIn
   returnA -< ((sinOut, fOut), sinf-val)
```

Listing A.4: An Arrow in Point-wise Style (With Syntactic Sugar).

Here, we see the special *proc* command that creates an Arrow procedure. The incoming value to the Arrow is the next portion of the syntax. Arrows can have only a single input, so the values here are paired. Finally, the *do* notation is adopted from the Monad structure. The body of the *do* is made of Arrow computations. The input and output are specified on each line, with the "shaft" of the arrows (the visual representation that looks like this: $\prec\!\!-\ -\!\!\prec$) being instances of the Arrow type class. The Arrow f and the lifted sin function are the Arrows used. The *returnA* is a special Arrow that applies the identity function to its input and returns the value as the output of the Arrow. The finally addition of the point-wise style is the *rec* command. This indicates that an ArrowLoop is used and what portion of the code fragment contains the loop.

In certain cases, the point-free style of arrow programming is still very important. In particular, it can be advantageous to use in defining the functionality of an Arrow in the "shaft" of another procedure. The value *sinf−val* in Listing A.4 is calculated using two functions in a very simple point-free way.

The Arrow extension to Haskell is not yet an official part of the Haskell standard, though it will purportedly be part of the next standard. The Arrow concept is introduced as a type class into Haskell, so it is possible to use in any compiler. The point-wise notation can be transliterated into pure Haskell (i.e. the point-free style). A special arrow preprocessor was developed to achieve this, converting *.as* files *.hs* files. This method is still required for usage with two of the major compilers, the HUGS compiler and the nhc98 compiler. The most widely spread compiler, ghc, has since incorporated the arrow preprocessor into the compiler and is activated with single switch. This lends some validation to use of the point-wise syntax and assures easier and enduring usage.

A.4 Yampa: Arrowized Functional Reactive Programming

This section serves to introduce the Yampa FRP framework. Yampa was selected as the basis system for the dissertation based on its unique abilities and its possibilities for extension. The presentation of Yampa is divided up into two parts. In the first part, the Yampa implementation method is described. In particular the mechanisms Yampa uses for the behavior simulation are detailed; those details are important to understanding the FRP extensions presented in Chapter 8. The second part of this section presents the basic functionalities that Yampa has built into it.

A.4.1 Implementation Details

The details of how Yampa is implemented are important to understanding how many of its properties are made possible. These details are also important in understanding how the different functionalities present and those developed in this dissertation function. As such, portions of this discussion are in-depth and dependent on an understanding of Haskell and particularly Arrows; with an understanding the basic concepts, as presented in the last section, the implementation details should be comprehensible. However, the

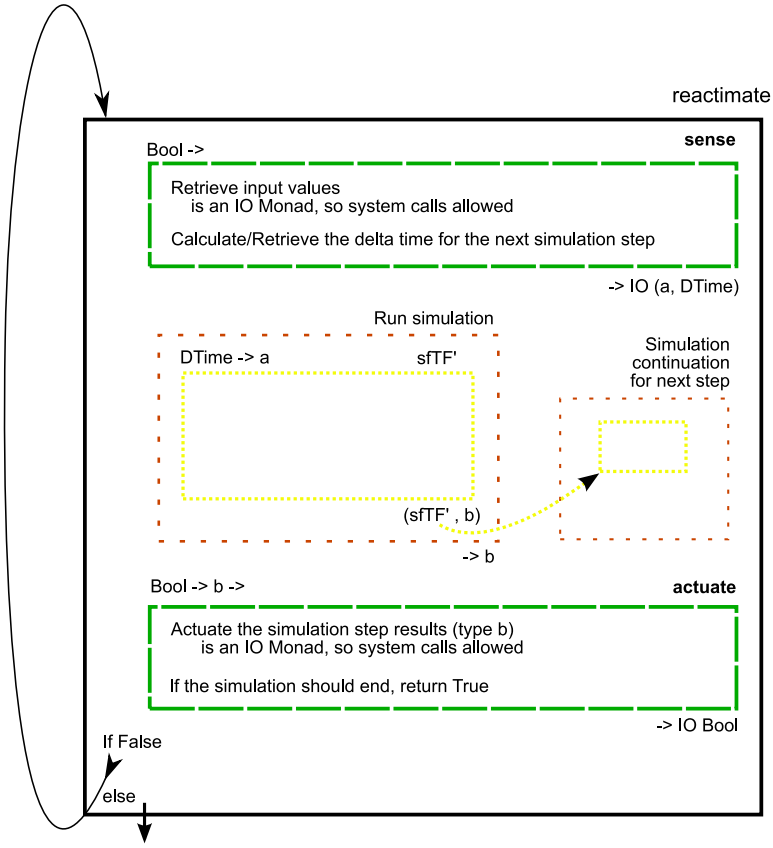


Figure A.3: The diagram shows the simulation loop of Yampa, as performed in the *reactimate* simulation kernel.

detail will reach beyond this basic understanding, enabling a critical understanding for those with a deeper knowledge of Arrowized Haskell. Two core portions of the Yampa implementation are explored in this sub-section, the kernel and the underlying core to the behaviors. The functionalities provided by Yampa, including the reactive nature and actual processing of behaviors are introduced in the next sub-section.

The simulation kernel that drives the Yampa system is the *reactimate* function. This Haskell function is designed to run the simulation only returning when the complete simulation is ended. There is an alternate to run this the *react* each step from Haskell, though there are no examples of its usage. The simulation loop of *reactimate* is shown Figure A.3. The reactimate function takes four inputs, the initial input (as an IO monad, so it can be called from the main Haskell function), the sense function, the actuate function, and the SF to run. The sense function is a Haskell function responsible for retrieving the new input value and the time delta for each step. The actuate function is responsible for distributing the output value of each step and returns IO Monad tagged with a bool indicating whether to continue or not. Finally, the SF is what the simulation actually does, typically containing a complete hierarchy of SFs.

As a functional implementation, the Signal Function is defined to depend only on the inputs to the SF up to that moment in time and time itself. Time is restricted to being non-negative in the Yampa implementation. These fundamental rules are the cornerstone of achieving a valid system. A critical aspect the SF implementation is that it is dependent on not only the momentary inputs, but may also include a feedback loop. This is enabled by the SF implementing the ArrowLoop type. When the functionality is used, the signal is dependent on the momentary inputs and the inputs all previous steps. This is an important concept, as it implies that the behavior is the culmination of all inputs from the beginning of simulation time. For instance, in the context of DIVEs this means that a behavior like Boids that is reactive to continuous inputs is the summation of the entire time, not just a single moment. This is critical for functionalities like undo (see Section 9.4 for details).

The basic type, SF, is a data type that is an instance of the Arrow type class and also of the ArrowLoop class. The importance of the Arrow type class is that it allows handling the behaviors as computations. For the FRP programmer this has little impact, but on the implementation side it has significant impact. Using the arrow type as computations, the "space time leaks" of previous versions of FRP are removed without sacrificing FRP's expressibility and power. Details of how this works are documented in Courtney's dissertation [Cou04] over Yampa and beyond this discussion. The ArrowLoop functionality is necessary so that state can be held, in particular for behaviors based on integrals.

While Arrows are the key to the SFs stability, the implementation detail that is the core to making SFs function is a continuation based implementation. Using this approach allows the SF to be extremely flexible and enables the possibility to build many of the functionalities required in this dissertation. The continuation is built by making the SF a transition function; the transition function transforms an underlying implementation function to a new (continuation) function and has a side effect of producing an output at every step. The continuation implementation is too complicated to thoroughly explain here. However, as no published documents truly delve into them at the level required here, a bit of an explanation follows.

SFs are complex data structures built on continuations. Before the SF can be executed it must first be initialized. An SF in the uninitialized state is in a "frozen" state, meaning it cannot be executed. After initialization, priming it with initial input and delta time values, it is said to be a "running SF." So, a running SF is a transition function that receives an input value and a time delta. It produces two outputs, a new running SF and an output value for the behavior it implements. Since the SF is simple a data object that contains the transition function to run, the Yampa kernel is responsible for the execution of the behavior. The simulation kernel "runs" the SFs by providing input and time and then handling the output. However, before exploring that functionality, a closer look at the implementation is necessary. Listing A.5 shows the Haskell code and a textual explanation follows.

```
-- Signal Function data type definition
-- has a variable, the sfTF (Transition Function)
-- that takes an input and returns the ``Transition''
data SF a b = SF {sfTF ::  a -> Transition a b}

-- The transition is a ``running SF'' (a continuation) and the output
-- of the current run
type Transition a b = (SF' a b, b)

-- The ``running SF'' type SF'
-- actual behavior implementations live here
-- is a function that takes a delta time an input a
-- and returns a new Transition (i.e.\ the continuation)
data SF' a b
    = SFConst {sfTF' :: DTime -> a -> Transition a b, sfCVal :: b}
    | SFArr   {sfTF' :: DTime -> a -> Transition a b, sfAFun :: a -> b}
    | SFTIVar {sfTF' :: DTime -> a -> Transition a b}
```

Listing A.5: Signal Function Type Definition

The *SF* type is a simple data type that implements the *Arrow* and *ArrowLoop* type classes. It defines a "frozen" signal function. It is the data type that users of Yampa deal with, although even then only through special functions. These functions mirror those existing for Arrows; for instance, the main function developers deal with is *sfArr*, the lifting function. The data the SF holds is a "transaction function" that initializes the SF. It returns a Transition, the static part of the continuation process.

The *Transition* type and the *SF'* data types are the core to continuation process. For the length of the time that simulation of the behavior occurs, the kernel uses these types to simulate everything. The *Transition* is how the behavior lives when it is not executing. It contains the continuation, the running SF (*SF'*) to execute the next simulation step, and the result of the step just processed. The heart of the behavior is the function contained in the *SF'* data. Three specializations of the running SF are possible, but the core of them is the special *sfTF'* function. This function takes the time and input and returns a *Transition*, i.e. the continuation and behavior result for that step. The kernel saves the continuation to use in the next step and provides the output to the next simulation step.

The *SF'* has three specializations. The *SFTIVar* specialization is the general case. The other two provide speed ups when simulating special cases. SFConst is a running SF that has a constant value and therefore not dependent on time, returning a value immediately without calculation. SFArr contains a pure Haskell function (i.e. the lift functionality *arr*) and is, therefore, also not dependent on time. Both these return the same data as the continuation. Finally, the SFTIVar is the catchall that is time varying. A new continuation is calculated every step as it is time dependent.

Although only a single functionality provided by Yampa uses this possibility, the continuation implementation makes it possible to take a "running SF" and make it back into a SF data type. Courtney refers to this as freezing the SF and aptly names the function *freeze*. In essence this takes a "snapshot" of an SF, capturing its state. The kSwitch presented in the next sub-section is the only functionality provided in Yampa which utilizes this capability and the freeze functionality is not exported.

Having introduced the continuous behaviors, the reactive nature remains unclarified. The basic discrete and reactive components of Yampa build upon the *Event* data type. This type follows the paradigm of defining an event that is a possibility and the occurrence of that event. This is programmed in the same way as the Haskell Maybe type. The data is either the *NoEvent* or an *Event a*. Here, the second important part of event occurrences in Yampa can be seen. When events occur they can have data attached to them. This can be used in various ways and there are many functions in Yampa for dealing with events and the data that is attached to it. The reactive nature based on the events will be introduced in the next subsection covering the built-in functionalities of Yampa.

A.4.2 Built-in Functionalities

Yampa is completed by a collection of functionalities that truly are what makes it a hybrid system solution. The core functionalities presented in the previous sub-section are the programming details of how Signal Functions are created and how they are imbued with the properties that make them capable of implementing everything needed for DIVEs. In this sub-section the functionalities provided in Yampa that make it part of the FRP family are introduced, i.e. what makes up the Functional and Reactive components. Yampa's "functional" functionalities are presented in the first part. The provided methods for composition of the built in functionalities into higher level behaviors are introduced next. Then, the event system is introduced. Finally, the "reactive" functionalities in Yampa are presented.

When considering the SFs of Yampa, it is important to note that SFs are not defined by the developer directly. Instead special functions, like the arr lifting function introduced previously, are used. This can create some confusion when discussing Yampa components. Although the integral introduced above is a Haskell function, its return value is an SF. That SF is the environment that is then simulated by the Kernel.

"Functional" Programming Built-ins

The FRP style of programming is based on a building block nature. At the lowest level are primitive SFs that lay the foundation for time based behaviors. Over these are the SFs that enable the building of complex systems from those basis functionalities.

Yampa provides a number of SF primitives as building blocks. The most basic of all SFs create non-time based SFs. The *arr* lifting function changes a Haskell function into an SF. The *identity* function creates an SF that just passes a value through. The *constant* function creates an SF that always returns the same value. Beyond these trivial SFs, two classes of functionalities are supported: calculus based and control system based functionalities.

The most basic functionality Yampa provides for continuous time behaviors is the *integral* SF. Using the integral functionality, the user easily defines continuous time behaviors using higher order functions. Yampa currently uses the "rectangle rule" for

integration, though other integration methods were created [Cou04]. The standard integration essentially has a step delay incorporated into it. An extra imIntegral provides immediate integration. Finally, the inverse is available also in the form of a derivative function. Although, not used in the examples, this functionality is useful for work with continuous input.

A single special function exists that uses the integral, the *time* SF. It returns the simulation time since the start of the SF. It is important to note that this is the "local time" of the SF not the complete simulation; therefore, *time* is actually an alias of *localTime*. The time SF is defined as the integral of a constant.

The control system inspired behaviors show some of the possibilities of what can be built and are the demonstration tools from Courtney on composing higher level functionalities. The most basic of these functionalities is the delays. These are simple SFs that delay the values one step. Of particular importance is the *iPre* SF, which delays and is pre initialized. This is important for SFs that contain loops (recursive values) so that they are well defined. Other functionalities include accumulators, hold, and track and hold behaviors that are familiar to control systems.

Behavior Composition

Much of the power and flexibility of FRP and Yampa comes from the composability of behaviors to make complex systems. As the SFs are based on Arrows, the standard method of connecting the computations together forms the basis of the system (see Section A.3.2). Using the syntactic sugar variant programming style, this reduces primarily to the use of the composition operator $\ggg$. This is simple to use to pipe the output of one SF to the input of the next. Yampa provides several additional composition possibilities of importance.

The simplest of these is the *parallel composition* SFs. This allows joining similar SFs into collections. There are two basic forms to parallel composition in Yampa, differing in how the input is delivered to the composed SFs. The basic form is *broadcast*, where each SF gets the same input. Functions using this distribution method add a *B* suffix to the SF name. The second form has a *routing* function that determines what input is delivered to each function. The *par* and *parB* SFs implement the parallel compositions on collections. Parallel composition is also combined with other behaviors, particularly the reactive switches presented below in Section A.4.2. In those cases, the *p* prefix is added to the SF name.

Events

As explained previously, Events form the foundation of the reactive system that is presented in the next part. However, Events are simply data. The source of an event can be anything, but special functionalities are provided and of interest. These functionalities handle the generation of events with respect to time. Examples are SFs that trigger

an event after X seconds or at time Y. These create simple and powerful mechanisms for timed systems.

Time related Event generation SFs include:

after generates an event occurrence, with a specified value, after X seconds then never generates an event again,

afterEach takes a list of intervals at which events should occur along with their values,

repeatedly generates events every X seconds with the same tagged value,

occasionally generates an event occurrence approximated every X seconds. Fuzzyness to the timing is achieved by using a random number generator (the provided *noise* function) for the difference to the given time.

Other useful Event SFs that are included are a series of edging functions. These SF trigger events based on different criterion. For example the edgeBy SF uses a specified function to test for the occurrence of a value. Event suppression is sometimes also required and a series of behaviors exist to handle these cases.

"Reactive" Programming Built-ins

Support for reactive action is provided at two different levels, though the FRP community only considers one of them. The implicit reactivity that is built into the FRP system is that of interactive behaviors based on its inputs. This could be interactivity in an external system like a GUI in regards to "drag'n'drop" style behavior or it could be interaction between behaviors within the simulated environment. The reactive nature that is spoken to in the FRP title is that the system can react to event occurrences.

The main tool Yampa provides for interactivity and for flexibility is a series of *switches*. Switches, in their basic form, react to an event occurrence by changing the active behavior to a new behavior. What this means is easily seen in the diagram of Figure A.4. Naturally, the behavior that is running in the switch can be any SF. This means a composite functionality can be used, but also that hierarchies of switches can be built. Switches are available in three basic variations, where each has variations for parallel composition and delayed processing. These switches at a conceptual level are:

switch basic switch, switches on the first event occurrence, with the switch event being generated by the SF that is running

rswitch a externally triggered switch, where the event is tagged with the new SF to switch into

kswitch is a switch triggered through an external function. Using a "call with current continuation," which captures the current SF state, the switching function can either continue the previous function or switch into the new switch.

The switches are of particular importance, so their exact functionality is investigated individually.

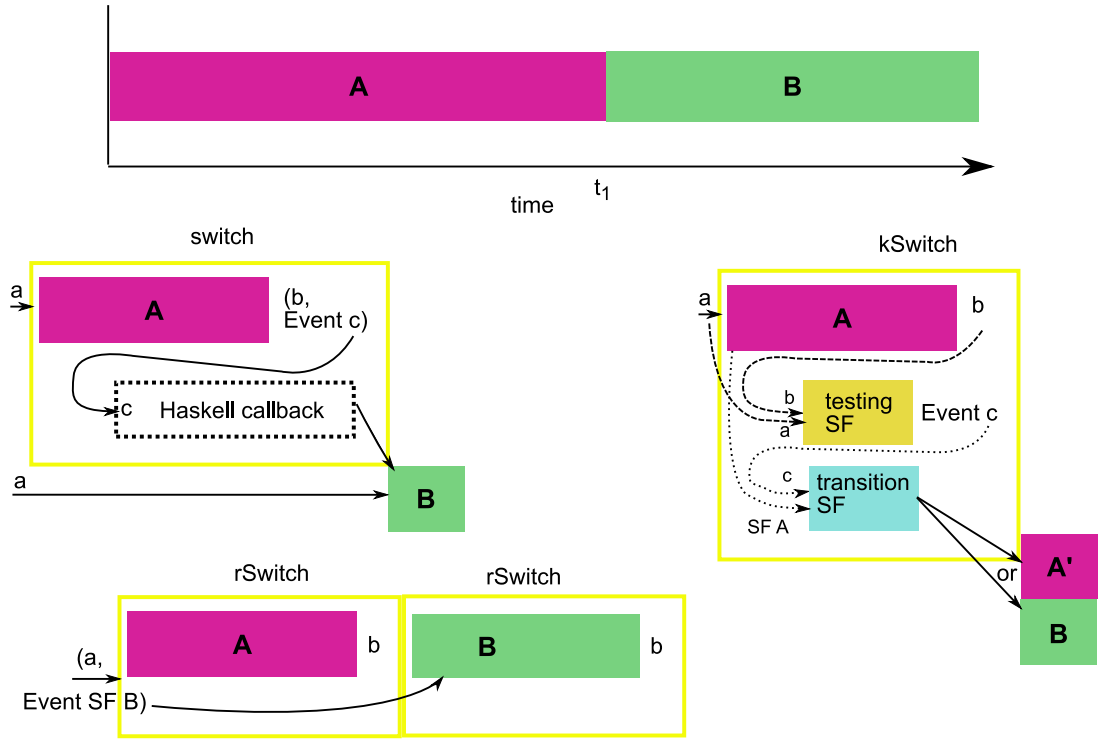


Figure A.4: The diagram shows the Yampa Switching functionalities. SF *A* runs until the switching event occurs, at which point SF *B* starts.

The **switch** is the simplest and easiest to understand. The *switch* SF generating function takes two parameters. It has a signature of:

```
switch :: SF a (b, Event c) -- initial SF
       -> (c -> SF a b) -- on event, callback 2 generate new SF
       -> SF a b
```

The first argument is the initial SF. It takes the input as usual. However, the SF must return both the value of the behavior to be returned and an Event. On the event occurrence, the second argument to the *switch* function is used to generate the new SF. This is a callback function, which receives the tagged information in the Event that was through and generates the SF that will be switched into. It is important to note that the new SF is no longer wrapped in the switch environment. Often the new SF is another switch.

The **rswitch** is the "recurring" switch, though the major difference here is really the location in the simulation where the event and new SF are generated. In the switch, the event was generated by the encapsulated SF. The *rswitch* receives the event from the outside and the tagged value of the event is the SF to change into.

The type signature of the rSwitch is:

```
rSwitch :: SF a b -- initial SF
           -- SF to change into is provided in tagged event input
           -> SF (a,Event (SF a b)) b
```

Though this is named the recurring switch, the essence of it implementation is an external switch. This method has essentially two impacts on the programming. One is that the decision to change has to occur from the outside, meaning it only has information about the input to the behavior, not the output or any internal information. Externally generated events are handled well with this method. The second difference is that the SF to change into has exactly the same information. The next SF has to be specified without access to the internals of the behavior or its output. The standard *switch* could provide internal information to the switching structure. For a recurring switch - simple reloading of a SF that is specified without context information - the rSwitch is optimal.

The **kSwitch** introduces a novel functionality. Similar the standard *switch*, a special function creates the new behavior to switch into; however, the *kSwitch* introduces the ability to decide to continue the original behavior. This is achieved by using a "call with continuation" that is performed with the freeze functionality described above. The *kSwitch* is different in that it also has an external event generation method. After the initial SF runs (plain like the rSwitch) a second, special SF is run which determines if an event has occurred. This generates the event The *kSwitch*'s type is:

```
kSwitch :: SF a b -- initial SF
           -> SF (a,b) (Event c) -- testing SF
           -> (SF a b -> c -> SF a b) -- transitioning function
           -> SF a b
```

The initial argument provides the initial SF to run as usual. The second SF tests for an event occurrence. There are a number of things to note with this functionality. This is an SF in itself, meaning it can have state. It is also of importance that the input the testing SF is the input as well as the output generated at that step. The final input argument is the function which generates the new SF on an Event occurrence. Here, the Haskell function receives two parameters, the frozen initial SF and the value of the event generated by the testing SF. This event value is used the same as in the case of the standard *switch*. This function is then allowed, depending on the value of the event to either continue the SF or to generate a new SF. A final note is that when the event occurs, regardless of whether the initial SF is chosen or a new SF is created, the replacement runs outside of the kSwitch environment, i.e. ending the switching capability.

The kSwitch is the most complicated of the switches. The combination of abilities it is equipped with make it a bit complicated to use. Additionally, the continuation functionality of the kSwitch is not strictly necessary; the only information that the transitioning function has is the event tagged value, which is generated in a SF that receives only the input and output values. All of this could be done just as easily inside

the initial SF using a standard *switch*. Several of the ideas of the kSwitch are use in Section 8.4, where specialized transitioning switches are created.

A.5 FRP in Use

FRP has largely been an academic exercise to this point. However, its development has always been application informed. Its origin is as a DSL for programming computer graphic animations [EH97, ESYAE94]. Animation has remained a recurring application area over the course of FRP's development [Ell98, Ell99]. FRP has also been used for other application areas. Examples include robotics [PHE99, PHH99], games [CNP03, Mun05], Graphical User Interfaces (GUIs) [CE01, Cou04], embedded control systems [NCP02], and music and dance generation [HH03, Hud00]. A short introduction to two of these applications, robotics and games, indicates some of the power and flexibility of FRP.

One of the earliest and potentially most interesting application areas explored, starting with the FRP generation of systems from Yale, is robotics [PHE99, PHH99]. The developments are best known by the moniker FRob (**F**unctional **Rob**otics). The synergy between VR and robotics makes this an interesting system, as both are concerned with a hybrid system of continuous input and outputs and discrete events.

Another relevant test bed has been games, particularly as an application for Yampa. Attempts to program games showed some of the limitations of the previous versions of FRP and was a part of the motivation to reinstate the flexibility of early FRP systems in Yampa. The example programmed in Yampa's development was a "Space Invaders" clone [CNP03]. An interesting part of that development is the flexibility achieved in the system. All of the invading aliens are added to the system dynamically, generated at random times by built-in noise functions. The other interesting part of the space invaders implementation is that the number of enemy ships is controlled through a dynamic collection of ships. This demonstrates Yampa's run-time changeable nature.

A more recent usage of Yampa for game development was in the development a First Person Shooter [Mun05]. Here, a primitive First Person Shooter style game was developed as part of a Master's thesis. In this case, the application was programmed by a student outside of the FRP development community, showing that is possible to use "outside of the lab."

Appendix B

Visual Programming

The development of the Arrow extensions to Haskell are a key part of the success of the Yampa version of Functional Reactive Programming used in this dissertation (Appendix A introduces Haskell, Arrows, FRP, and Yampa). However, the Arrows bring with them difficulties. The complication the syntax of the Haskell language through the Arrows extension, even with the very important pointed "syntactic sugar," makes the functional style of Haskell less clean. Since the context of this dissertation is in a real-time graphics area, the functional language style is foreign in itself. In this Appendix, we present an extension of the Arrowized Haskell language to allow the author of the environment to program the more complex aspects in a visual manner.

Graphical representation has a long history in software development [Sch98]. The usage of the graphical representations in programming ranges from tools to help gain understanding and insight into the structure of existing code bases to systems that enable programming through the use of graphical representations. The latter is often denoted as *Visual Programming*. Visual Programming techniques exploit the fact that people learn to understand images much earlier than text [Sch98]. Visual Programming can be divided into two classes, Visual Programming Languages and Visual Programming Environments. Where Visual Programming Languages builds a new language, a Visual Programming Environment (VPE) builds upon a conventional programming language. The visual program is translated into the underlying language and converted to executable code by conventional means.

In the explanation of the working of AFRP and Arrow code we have frequently used graphical representations to help explain their workings, e.g. Figures 9.3, 9.5, and A.2. This is also something found in most explanations of Arrows as well as common to Courtney's dissertation on Yampa [Cou04]. Additionally, we found ourselves, in developing this work, performing the actions of drawing diagrams in order to not only understand how things worked, but also to understand how to program new problems. These graphical representations suggested that the Visual Programming Environment would be advantageous.

A few attempts to create visual programming of functional languages are known to the authors [Han02, Kel02, Ree95]. The works have all focused on the Haskell language. In his dissertation [Ree95], Reekie focuses mostly on representing pre-existing code in a visual environment to obtain a better understanding of the code. His approach is based on the view of the system as a data-flow system, a complementary view to our work. Kelso, in his Dissertation [Kel02], develops a visual syntax for visual programming of functional languages. Though, his approach is largely held abstract from a specific language, it is implemented using the Haskell language. Kelso explicitly excludes the components of the Haskell language of interest in this paper, mentioning that they are well suited for visual programming techniques. Instead, he focuses on "standard" pure functional programming. Hanna's Vital environment uses Haskell for visual spreadsheet programming [Han02]. Haskell expressions define the layout and values of the spreadsheet components; Vital is a live interactive visual environment. Finally, Dami and Vallet have explored visual programming techniques for higher-order functional programming [DV96]. Although their work was for didactic purposes, it is the only works to focus on higher-order programming.

This chapter reports on the development of a Visual Programming Environment for Arrow based Haskell programming. The system created is held general to the needs of Arrow based programming and also the needs of FRP/FRVR programming. The work summarized here was performed by a German Diploma student, Piotr Szal, under the guidance of the authors. The complete work, in German, is described in Szal's Thesis, "Visuelle Entwicklungsumgebung zur Erzeugung von Haskell AFRP Code" [Sza07].

B.1 Visually Programming Signal Functions

The Arrows framework naturally matches the visual programming paradigm. The areas with the highest success in visual programming have been data-flow systems, something that is afforded by the computation concept that Arrows are based on. Even Kelso noted this in [Kel02]: "Function composition pipelines and Monads are candidates for special syntactic support in the (Visual Functional Programming Environment)."

Our motivation is to simplify the programming of systems based on Arrows. The Visual Programming Environment (VPE) simplifies programming in two main areas. The conceptual understanding of the code, either being inspected or written, is helped via the graphical representation. This provides for better speed in programming, producing good code, and is easier to use for non-expert users. The second place that the VPE helps is in dealing with "plumbing" issues. Here, the difficulty lies in gaining a conceptual understanding of how to do this with Arrows, in getting the syntax right, and in the tedium of coupling values together - which can become very large and complex.

In this section, we step-wise introduce our visual programming environment. First, an overview of the Arrow paradigm and its application are given. Second, we introduce how Arrows can be visually represented, without discussing system details. Thirdly, this

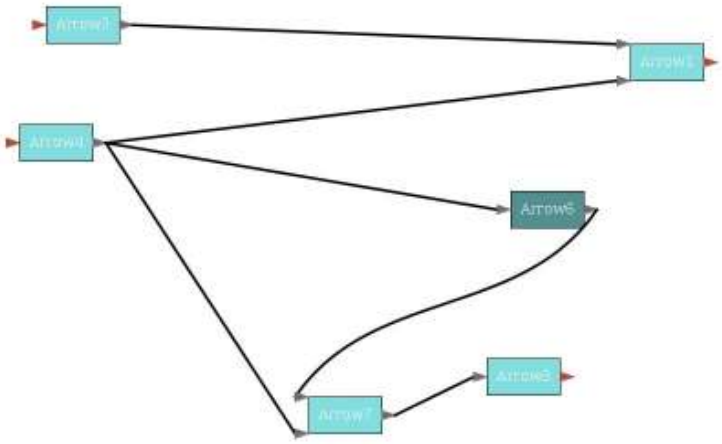


Figure B.1: A Network Formed from Primitive "Boxes"

high-level view of visual representation is implemented in a concrete system, explaining how each component is implemented in our system. Finally, the fine implementation details, such as the algorithm for creating the code, are given. Throughout this section, we focus on Arrows without considering their use context in systems that build on them. In Section B.5, we expand upon this system to support the extensions required for programming in real applications.

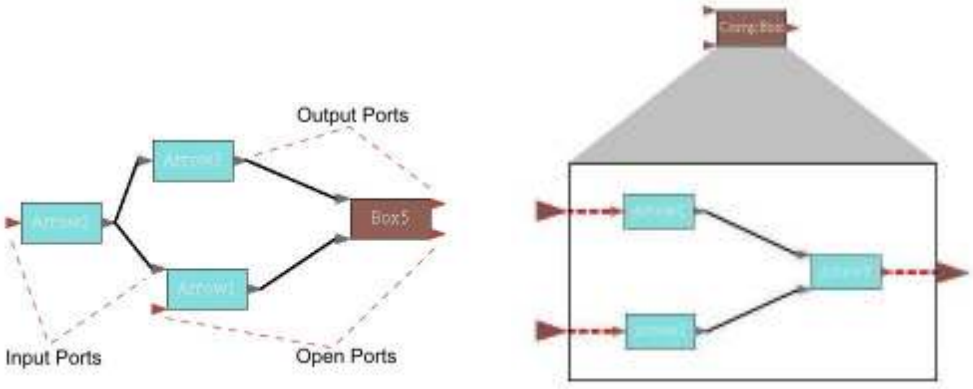
B.2 A Visual Syntax for Arrows

For the visual representation each Arrow can be thought of as a Black Box, which transforms the input signals into the output signals. The connection of the Arrows so that the input of one Arrow becomes the output of another Arrow can be visually shown as a connection between the Black Boxes [DV96]. This is also the classical data-flow view of the system, illustrated in Figure B.1.

In our Arrow environment, there are two possibilities for forming these boxes. A box can be lifted primitive (Haskell) function or contain a complex combination of other boxes. The compound boxes describe a higher-level functionality, formed out of both primitive boxes and other grouping boxes.

The most basic Arrow is defined by a *primitive box*, an encapsulation of a Haskell function. This is directly the Arrow lifting function, *arr*. The primitive Box cannot be broken down further on the level of Arrows - although the Haskell function could naturally be composed of any number of sub-functions. The primitive box is the basic building block of the VPE.

Each primitive box has input and output *Ports*. Arrows are defined to take a single input and returns a single value. In the textual programming environment, through the



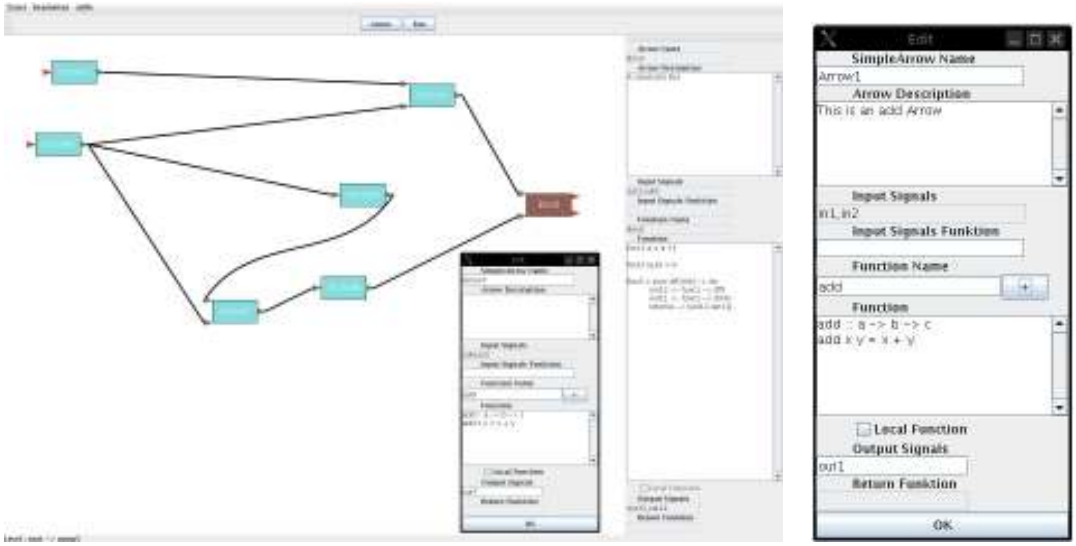
(a) Ports of Boxes (b) Internal Environment of a Compound Box

Figure B.2: Basics Components of the VPE

use of tuples, the user can simulate having multiple inputs and outputs. The values have to be manually "routed" in and out of tuples by the user. This approach is difficult in a visual environment. Adding extra functionality to insert and extract values from tuples would pollute the visual syntax and add unnecessary complexity. Instead, an expanded set of *arr* functions are used: functions, *arr*2 through *arr*5, as in Yampa [Cou04], are used. These specialized lifting functions take Haskell functions with 2 to 5 inputs and return an Arrow that receives a single input tuple with the corresponding number of objects.

In the visual environment the inputs and outputs of the Arrows are represented by special *Ports* on the boxes, as seen in Figure B.2(a). The number of ports correlates to the number of values in the tuple instead of the single Arrow input/output. An input port can have precisely one connection into it. An output port can, however, be connected to any number of input ports. Ports that are not currently connected in the visual environment are denoted as *Open Ports*. These Open Ports will be of importance in the next step, compound boxes.

Beyond the primitive box, a second type of box is required. This grouping or *Compound Box* is the combination of other boxes into a procedure, as introduced by Paterson [Pat01]. The compound box is built from both primitive boxes and other compound boxes. Figure B.2(b) shows a compound box in our visual system and the internals of the compound. As mentioned previously, an Arrow can have only one input and output value, where this is often a tuple of values. Similar to the primitive solution, the visual system can assist in the case of the compound boxes, by automatically performing the pairing. Compound boxes can, however, have multiple output ports in addition to input ports. The number of ports for a compound box is determined solely by the number of open - not connected - input and output ports in the drawing space that defines it. Figure B.2(b) shows a compound box and the drawing space, where two open input ports and one open output port are present.



(a) The Arrows Visual Programming Environment (b) code dialog

Figure B.3: The Arrows Visual Programming Environment: a) shows the program during use, b) shows the Editing Window for Haskell code. It allows the underlying function to be specified in arr primitive, names given to the function and inputs, etc.

In contrast to other visual programming of functional languages [DV96, Kel02], cycles in the data-flow can be allowed by using the ArrowLoop extension. ArrowLoop provides provisions for a feedback loop, specifically for static data. In the VPE, the user creates loops through the connections they build. The implementation details of loops are explained in Section B.4.3 and the visuals can be seen in Figure B.5(a) found there.

B.3 The Arrows Visual Programming Environment

Based on the visual components developed, a VPE is created for Arrows programming is presented here. Explanations of how all of the components are used and incorporated to write a program is provided. Each of the main components is explained in turn, focusing on how they are implemented in our system.

The developed visual programming environment is divided into four areas, as seen in Figure B.3(a). On the upper toolbar are buttons for the creation of new primitive and compound boxes. On the right hand side of the GUI is an area showing the details of the currently selected box, including the underlying code. The main part of the environment is the drawing interface. There, the boxes are positioned and edited, and ports are connected. Finally, a status bar at the bottom provides visual feedback in regards to the compound boxes.

B.3.1 Primitive Boxes

When a new primitive box is inserted, it appears in the drawing space. To begin with primitive boxes are empty templates; they contain no Haskell functions. The boxes can be positioned freely within the drawing space using standard drag'n'drop methods, though overlapping boxes are prohibited. The most basic next step is to edit the primitive box.

The primitive box is edited by opening the editing window, as seen in Figure B.3(b). In the editing window, the user can specify the underlying pure Haskell function and various attributes. A name can be given to the box or an icon can be specified. Such iconic representations can be an advantage for easy understanding of the functionality of the box. For instance, an $\int$ symbol is very appropriate for one of Yampa's core functionalities integral, which performs precisely that action. A textual description - a Haskell comment - can be assigned to the box. The input ports can be named through one of two mechanisms. When the port has not been connected, the user can name the input signals in the *InputSignals* field. When the input port is connected to an output port, it receives the output port's name. The output port names are defined in the Output Signals fields. For all ports, the names are displayed with tooltips on a mouse hover over the port.

In the *Function* field the Haskell function to be lifted is filled in, including the type definitions. This can be seen in Figure B.3(b). Three additional functionalities make this more robust. The function can be defined to be either defined locally (a where definition at the level of the procedure definition) or globally. If the function is defined globally, this provides the possibility for it to be used in more than one place. The + button found to the right of the Function Name field allows the user to select an existing global function. Finally, a function that already exists in an external Haskell module can be used. To do this, the user need only include the function definition. This is necessary, as the function definition is used to determine the number of ports the box has and for type checking. For instance, after filling in the primitive box as in Figure B.3(b), the box will have two input ports.

B.3.2 Compound Boxes

Compound boxes can be created through one of two methods. The first method is that the user inserts a new empty compound box to the drawing space by using the button on the upper taskbar. The empty compound starts with no inputs or outputs. Editing of the box happens through a slightly different mechanism, which is special to the visual environment. A compound box is an Arrow procedure that is specified in another drawing space. The compound boxes can be thought of level-wise and is implemented in this manner in our system. To edit the compound box on has to "enter" the compound box, through the "zoom in" command. So that the user can keep their orientation, current level of editing, within the layers of compound boxes, is displayed in the status bar on the lower portion of the interface.

The second method of creating a compound box is by creating the program fragment and, then, extracting out. The drawing space contents define the new compound box and the interface moves up a level, where the defined procedure appears as a compound box. Providing these two methods allows the user to program in two distinct styles, or a combination of them. By creating empty compound boxes and then filling them in, a top-down approach can be taken. A bottom-up approach is supported by building the functionality of the compound in the drawing field first and, then, extracting upwards. Naturally, the user is always free to move up and down levels and edit at any level.

As explained in the previous sub-section, the inputs and output fields of the compound are determined by the contents. In the interface, the compound box's input and output ports are automatically determined and generated by the system. This is determined by the number of open ports, input and output respectively, inside the compound. To incorporate extra input or input, one can simply enter the compound box and add the internal code. Ports left open will automatically be added to the compound box any time one zooms out.

The naming of the input and output ports is important for the usability of the visual programming system. This small help is indispensable for understanding what the inputs are, particularly in cases where there are more than one input. In the case of the compound boxes this is of utmost importance, as the system fixes the order of the inputs based on their position in the underlying drawing space. The names of the compound box's ports are made the same as those of the underlying port. They are sorted by Y position, ports appearing in that order. For example, in Figure B.2(b) the output port of the compound box will be named the same as the output port of right most Arrow.

B.3.3 Networks of Ports

With a combination of compound boxes and primitive boxes the user can build a program. The missing component to this point is the creation of data-flow in the program. This is performed through connecting the ports of the boxes. The creation of connections between ports is supported via the typical drag'n'drop metaphor. The connections between boxes is simply the $\ggg$ functionality, though our implementation relies on the Arrow do notation, so this is only true after the preprocessor of the compiler runs on the produced code.

The networking of ports introduces the second of the basic Arrow functionalities to the VPE. The final basic Arrow functionality is the $first$ function. The first functionality is intended to enable the sequencing of unrelated Arrows, so that the order of execution can be controlled. This is necessary in cases where side-effects should occur in a particular order. The $first$ command is simple, but rather difficult to grasp textually. Visually the $first$ command is much simpler to understand version, as demonstrated in Figure B.4.

Programming the first functionality in the visual environment is easily done by the user, though it is not as transparent as the arr and ggg functionalities. Using a

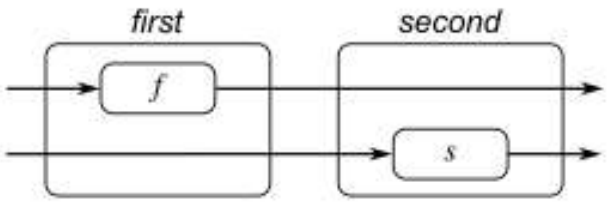


Figure B.4: The basic *first* functionality of Arrows shown with its pair *second*. Together, they enable ordering of unrelated Arrows with side-effects.

compound box, the user need only to use the first input to an Arrow that implements the function and the second input can be routed through a primitive that lifts the *id* function. A demonstration of such an implementation is included as part of a larger example in Section B.4.3.

B.4 Implementation Details

To this point the basic operation of the visual programming environment has been clarified; however, several details of the implementation have yet to be presented. These details are mostly contained to system level aspects. One aspect is the choice of language for programming the visual interface, described next. A necessary aspect of any programming environment is the ability to save, load, and export code. Here, the save and load functionalities deal with the visual program, while the export function is used to generate the Arrowized Haskell code. Each of these will be handled separately in this section.

B.4.1 Language of Implementation

The choice of language for the implementation of the visual programming environment is the most over-arching decision to be made. The program presented has been programmed purely in the Java programming language. The two main reasons for the Java implementation are code portability across operating systems and the prevalence of Java within the other visual functional programming projects, particularly Kelso's VFPE [Kel02].

B.4.2 Saving/Loading Visual Code

A natural necessity of the program is the ability to save the code and reload it. Moreover, the development of modules and/or code fragments that can be later inserted

is important. For this initial implementation, it was decided to provide two separate functionalities, one for saving/loading the visual code and an export of Haskell code.

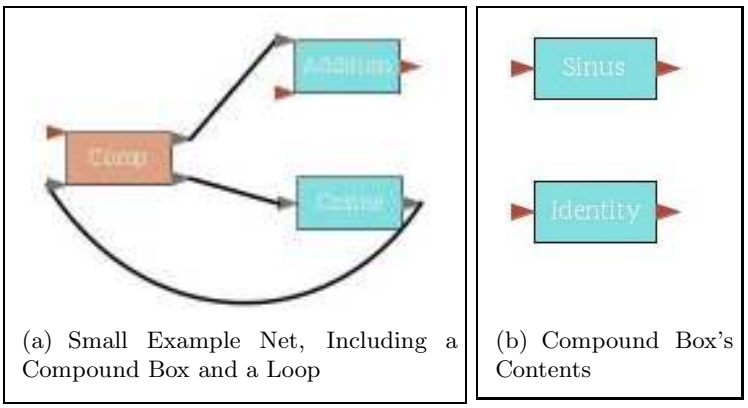
The presented implementation uses a functionality of Java that allows binary saving of sections of the environment in order to provide the load/save functionality. In this way, the precise visual environment is saved. For the loading functionality, we desired a more expressive system. The loading functionality is designed to allow the user to load the entire saved model or any subset of the saved functionality. When loading from a saved session, the user is presented with a tree of functionality that they can select from. The user can select any primitive or any level of compounds to import, which is added to the current drawing space. This allows a set of functionalities to be pre-programmed and the user can use those functions in their code.

B.4.3 Haskell Code Export

A description of the algorithm used for exporting the visual environment into Haskell code is provided here. At the highest level, a decision between generating point-free Arrows or "pointed" Arrows is required. While exporting direct as point-free Arrows would remove a step from the process, using pointed Arrows has a number of advantages and is ultimately the format chosen. The main advantage is that the generated code is more readable, when textually editing. The pointed Arrow code seemed more likely to be readable, particularly in a system, where users could leave auto-generated names. Additionally, since we are performing only rudimentary error checking in the visual language, the code will be needed for finding errors thrown by the compiler. Using pointed code means we still have the port names, etc., and can find location of the error in the visual environment again. With the inclusion of the pointed Arrows parser in GHC, the future of the notation seems assured. The final point is that the procedure of the pointed type is directly correlated to our compound boxes, making the implementation of the exporter much simpler.

The Primitive Box is the most basic functionality. As this is a single *arr* function, its output is fairly obvious. As mentioned in Section B.3.1, the VPE allows the user to handle multiple inputs to the Arrow, as if it were natural to Arrows. The extended *arr* functions transparently distributed tuples to the proper inputs of the Haskell functions; the collecting of the inputs into tuples has to be handled by the VPE. There are three possible placements of the source of the underlying Haskell function. The default is to place it in the global module space. The function can optionally be denoted as local and placed in a where clause. Finally, external functions can be used, in which case no definition is given in the exported code. An example of this can be seen in Figure B.5 and its code at the end of this section.

The processing of each compound box - and that of the highest level, which is viewed as a compound box itself - is where the majority of the work is performed. The first step in their processing is sorting the boxes on that level. Three separate lists are generated, a list of all the boxes, a list of boxes with an open input port (start boxes),



(a) Small Example Net, Including a Compound Box and a Loop

(b) Compound Box's Contents

Figure B.5: An example network, including: a compound box, loop, and three types of primitive boxes.

and a list of boxes with open output ports (end boxes). Each list is sorted by the X and Y (horizontal and vertical) axis coordinates, sorted in Y major order.

The next step handles the connectivity of the boxes. This is complicated slightly by the possibility of loops. This step works by sorting the complete list of boxes into an ordered list that is valid based on the dependencies asserted by the connections. The algorithm starts by moving all start boxes to the front of the complete list. From there, beginning with the top-most box the connection graph is followed, sorting boxes as it goes. Input dependencies are considered as the algorithm works forward, bubbling the dependencies further forward as required. During this process the algorithm has to detect loops, breaking out of cycles and marking that the compound has a loop within it. On completion of this step the complete list is sorted into a valid order, where dependent values are calculated before the box that requires them.

The final step is then to print out the compound using this list. The start box list holds a list of all open ports, for which an input tuple is created with proper names inserted for the values. If a loop was detected during the connectivity search, the rec keyword is inserted at the top level of the procedure and the Arrow is changed to an ArrowLoop. The algorithm, then, simply walks the list of boxes, having each box output its code. Each box becomes a single line in the procedure. The names for the values used in the pointed notation are assigned based on the connected input port that receives the value. Finally, the return values have to be collected, using the list of boxes with open output ports. The complete algorithm works with a bottom-up method, starting with the deepest embedded compound and working its way up.

```
comp :: Arrow a => (Float, Float) (Float, Float)
comp = proc ( inputSin, inputId ) -> do
        outSin <- sin -< inputSin
        outId <- id -< inputId
        returnA -< ( outSin, outId )

add :: Float -> Float -> Float
add a b = a + b
```

```
compEnd :: ArrowLoop a => a (Float, Float) Float
compEnd = proc (inputAdd2, inputComp1) -> do
        rec
                outCos <- arr myCos -< abs outId
                (outSin, outId) <- comp -< (inputComp1, outCos)
                outAdd <- arr2 add -< (outSin, inputAdd2)
        returnA -< outAdd
        where myCos :: a -> a
                myCos a = b
```

Listing B.1: Output Code for a Compound Box

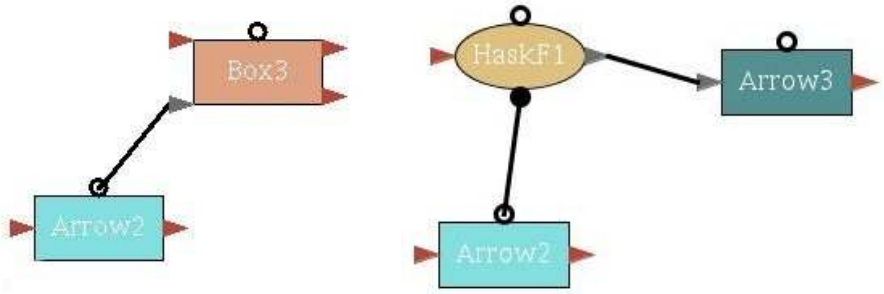
Figure B.5 and the related code segment above demonstrate many of the export functionalities. The code output is given for the compound box a level up from the editing space shown. The compound box in the drawing space, *comp*, contains two primitive boxes, one that lifts the *sin* function and one that lifts the *id* function, i.e. the $first$ functionality.

B.5 Visual Programming of Arrows as Higher-order Objects

The interface presented up to now is sufficient to create Arrows. However, Yampa's use of Arrows reaches beyond that of what is supported by the system to this point. There are a number of aspects to Yampa's usage of Arrows that makes programming with our current interface insufficient. Most of these revolve around Yampa's mixed usage of Haskell functions and Arrows and its use of higher-order programming. Yampa uses Arrows as computational units that are regularly passed around as objects. Even more of an issue is the reactive portions of Yampa. Much of this functionality is provided by pure Haskell functions. We can, however, through inspection of the functionalities provided in Yampa draw some of the similarities out, in order to find a solution to the problems presented by them.

Yampa's Event type and the related functions are useful as pure functions in many cases. In other cases, they cannot be simply lifted to Arrows. One such set of functionalities is the Haskell functions that produce an Arrow as their output. An example of this is the very frequently used $iPre$. The Haskell functions typically take constants, either at compile time or from the calling Haskell function. Another class of Haskell functions takes higher-order Arrow objects as their inputs and returns a new Arrow. Examples of these complex Arrow creating functions are the frequently used switches and parallel composition functions. The main complexity is that they typically take one or more Arrows as input as well as higher-order Haskell functions. Additionally, the switch functions are very often used in a recursive setup, requiring special handling.

A possible solution to these problems could be to re-implement the complete Yampa system in the VPE. However, this is not really optimal, as it is not really scalable to future uses and a loss of the optimized Yampa code is not desirable. We have developed a different solution to this problem by adding a rudimentary ability for the



(a) SF as Input to Another SF (b) SF as Input to a Haskell Function

Figure B.6: A pure Haskell function which returns a SF is represented in the VPE as an ellipse with standard ports and additional input ports for the Haskell function inputs.

user to create the pure Haskell functions in the visual environment. This is not a visual programming language for pure Haskell, but means to insert the function types and uses mentioned above. This requires three major components to be implemented: the Arrows as higher-order objects, Haskell functions that generate a higher-order Arrow object, and the ability to provide inputs to the Haskell functions, both constants and higher-order objects.

The simplest of these functionalities is using the Arrows (SFs) as higher-order objects that are given as inputs to other SFs or Haskell functions. To support this in the visual syntax, we add an extra "knob" to the visual representation that indicates that the Arrow should be used as a higher-order object. This "EyePort" can be seen in Figure B.6(a). The EyePort can be connected to the input of an Arrow, which takes an SF as input.

The next step is to include Haskell functions that produce Arrows as their output, as with the switch function. Figure B.6(b) shows how a pure Haskell function, which returns an Arrow, appears in the visual system. The elliptical form of the visual representation clearly separates it from Arrows, yet their use as an Arrow is allowed through the same port structure as the Arrows have. These ports are the ports of the returned Arrow from the function. Figure B.6(b) also demonstrates that the returned Arrow can be used as a higher-order object. The Haskell functions also require inputs. These inputs are given through special inputs found on the underside of visual object.

B.6 Discussion of the VPE

The resultant VPE seems to be well suited to building arrow programs. The data-flow style of the visual programming environment matches well with that of the arrow framework and is conceptually easy to follow. We believe that the case of arrowized code, and particularly Yampa's SF, is especially well suited to the visual programming method and the benefits are high, both for understanding of the code at a high level and for creating well structured code at this level. However, testing of the visual soft-

ware environment for code development, particularly for the non-programming experts, would be necessary to be able to support this. Although, other research indicates the benefit of visual programming, this is not yet tested in this case. To this point though, personal use has shown the approach to be beneficial.

One drawback of the current system, is that underlying Haskell code for basic functions must be coded in a simple editor outside of the visual environment. While the user must no longer deal with arrows and their syntax, having to still textually code inside of an otherwise visual environment makes mixture that may be irritating to some. One direction of interest is extending the system to include a system for visual programming of pure Haskell code. Kelso's VFPE [Kel02] would be a method would be an option. As both are written in the Java language, an implementation combining them would be straight forward. Unfortunately, Kelso's system was not distributed. Conceivably a combined system could be implemented by opening the visual Haskell editor instead of our current textual editor, both for the internals of the primitive boxes and the Haskell function ellipses.

Another area which would improve the current implementation is the replacement of the current method used for saving data. It uses a proprietary format and has a dependence on versions of the source code, such that saved data is not always upwards compatible with new versions of the software. A desirable solution would be to use Haskell code instead of a special format for the saving of the visual code. This extension would also allow easier import of functions - both pure and arrows - into the system. However, this is a complex problem, as going from arbitrary code to the visuals would not always create well structured visual code. A possible compromise solution would be to allow loading only of data generated by the VPE to be loaded instead of arbitrary Haskell arrow code. This could be supported by saving extended information in comments of each definition that would enable the system to properly recreate the visual code.

Bibliography

[ACHH93] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. **Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems**. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems I*, Lecture Notes in Computer Science 736, pages 209–229. Springer-Verlag, 1993. 36

[AD90] Rajeev Alur and David L. Dill. **Automata for modelling real-time systems**. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, LNCS, pages 322–335. Springer-Verlag, 1990. 35

[AD94] Rajeev Alur and David L. Dill. **A Theory of Timed Automata**. *Theoretical Computer Science*, **126**(2):183–235, 1994. 35

[AHJ+01] Robert S. Allison, Laurence R. Harris, Michael Jenkin, Urszula Jasiobedzka, and James E. Zacher. **Tolerance of Temporal Delay in Virtual Environments**. In *VR'01: Proceedings of the IEEE conference on Virtual Reality*, pages 247–254, 2001. 58, 59

[AHKV04] Christoph Anthes, Paul Heinzlreiter, Gerhard Kurka, and Jens Volkert. **Navigation models for a flexible, multi-mode VR navigation framework**. In *VRCAI '04: Proceedings of the 2004 ACM SIGGRAPH International Conference on Virtual Reality Continuum and its applications in Industry*, pages 476–479, New York, NY, USA, 2004. ACM Press. 65

[ALE03] Bernard D. Adelstein, Thomas G. Lee, and Stephen R. Ellis. **Head Tracking Latency in Virtual Enviroments: Psychophysics and a Model**. In *Proceedings of the Human Factors and Ergonomics Society's 47th Annual Meeting*, pages 2083–2087, 2003. 58, 59

[All91] James F Allen. **Time and Time Again: The Many Ways to Represent Time**. *International Journal of Intellegent Systems*, **6**(4):341–355, July 1991. 23, 24, 25

[Alu99] Rajeev Alur. **Timed Automata**. In *Proceedings of the 11th Internation Conference on Computer Aided Verification*, pages 8–22. Springer-Verlag, 1999. 35

[ANM97] Andrea L. Ames, David R. Nadeau, and John L. Moreland. *The VRML 2.0 Sourcebook*. John Wiley & Sons, Inc., 1997. 106

[Bar] Matt Barton. **Scorched Parabolas: A history of the Artillery. Game**. Armchair Arcade, On-line article. last visited: Sept 2008. Available from: http://armchairarcade.com/neo/node/427. 193

[BB05] Kristopher J. Blom and Steffi Beckhaus. **Emotional Storytelling**. In *IEEE Virtual Reality 2005 Conference, Workshop "Virtuality Structure", DVD of IEEE Virtual Reality 2005 Conference*, pages 23–27, Bonn, March 2005. 37

[BB06] Kristopher J. Blom and Steffi Beckhaus. **Making large, encompassing projects with individual, well-identifiable components work in VR classes**. In *"Virtuelle und Erweiterte Realität" 3. Workshop der GI-Fachgruppe VR/AR*, pages 37–48. Shaker Verlag, 2006. 211

[BB07a] Kristopher J. Blom and Steffi Beckhaus. **Functional Reactive Virtual Reality**. In *IPT/EGVE '07: Short Paper Proceedings of the IPT/EuroGraphics workshop on Virual Environments*, pages 295–302. EuroGraphics Association, June 2007. 218

[BB07b] Kristopher J. Blom and Steffi Beckhaus. **Integrating Functional Reactive Programming in a High-Level VR Framework**. In *Proceedings of the "Virtuelle und Erweiterte Realität" the 4th Workshop of the GI working group VR/AR*, pages 189–196, 2007. 140, 218

[BB07c] Kristopher J. Blom and Steffi Beckhaus. **Supporting the Creation of Dynamic, Interactive Virtual Environments**. In *VRST '07: Proceedings of the 2007 ACM symposium on Virtual Reality Software and Technology*, pages 51–54. ACM Press, 2007. 218

[BB08] Kristopher J. Blom and Steffi Beckhaus. **On the Creation of Dynamic, Interactive Virtual Environments**. In *In Proceedings of the IEEE VR 2008 workshop "SEARIS - Software Engineering and Architectures for Interactive Systems." 2008*. Shaker, 2008. 218

[BBH05] Steffi Beckhaus, Kristopher J. Blom, and Matthias Haringer. **Intuitive, Hands-free Travel Interfaces for Virtual Environments**. In *New Directions in 3D User Interfaces Workshop of IEEE VR 2005*, pages 57–60, March 2005. 64

[BBH07] Steffi Beckhaus, Kristopher J. Blom, and Matthias Haringer. *Concepts and Technologies for Pervasive Games: A Reader for Pervasive Gaming Research vol. 1*, chapter ChairIO - the Chair-Based Interface, pages 231–264. Shaker Verlag, November 2007. 64

[BC03] Grigore C. Burdea and Philippe Coiffet. *Virtual Reality Technology*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2003. 14, 15, 63, 87, 99

[BD07] Don Brutzman and Leonard Daly. *X3D: Extensible 3D Graphics for Web Authors*. The Morgan Kaufmann Series in Interactive 3D Technology. Morgan Kaufmann, 2007. 106

[Bec03] Steffi Beckhaus. *Dynamic Potential Fields for Guided Exploration in Virtual Environments*. Number 6. Fraunhofer Series in Information and Communication Technology, December 2003. 78

[BFP96] Salvador Bayarri, Marcos Fernandez, and Mariano Perez. **Virtual Reality for Driving Simulation**. *Communications of the ACM*, **39**(5):72–76, 1996. 16

[BG95] Jean-Francis Balaguer and Enrico Gobbetti. **Supporting Interactive Animation Using Multi-way Constraints**. In *Proceedings of the EuroGraphics Workshop on Programming Paradigm Graphics*. EuroGraphics Association, 1995. 108, 122, 227

[BH99] Doug A. Bowman and Larry Hodges. **Formalizing the Design, Evaluation, and Application of Interaction Techniques for Immersive Virtual Environments**. In *Journal of Visual Languages and Computing*, pages 37–53. Academic Press, 10 1999. 18, 20, 44, 48, 89, 91, 220

[BI06] George Baciu and Bartholomew K. C. Iu. **Motion retargeting in the presence of topological variations: Research Articles**. *Computer Animation and Virtual Worlds*, **17**(1):41–57, 2006. 28

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, London, 2nd edition, 1998. 228

[BJ98] Allen Bierbaum and Christopher Just. **Software Tools for Virtual Reality Application Development**. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer Graphics and Interactive Techniques: Course Notes*, New York, NY, USA, 1998. ACM Press. 102

[BJH+01] Allen Bierbaum, Christopher Just, Patrick Hartling, Kevin Meinert, Albert Baker, and Carolina Cruz-Neira. **VR Juggler: A Virtual Platform for Virtual Reality Application Development**. In *Proceedings of the Virtual Reality 2001 conference (VR'01)*, page 89, 2001. 102

[BK04] Steffi Beckhaus and Ernst Kruijff. **Unconventional Human Computer Interfaces**. In *SIGGRAPH '04: Proceedings of the 31th annual conference on Computer Graphics and Interactive Techniques: Course Notes*, page 18, New York, NY, USA, 2004. ACM Press. 62, 89, 94, 99, 100

[BKLP05] Doug A. Bowman, Ernst Kruijff, Joseph J. Laviola Jr., and Ivan Poupyrev. *3D User Interfaces: Theory and Practice*. Addison-Wesley, 2005. 13, 14, 15, 20, 58, 59, 62, 87, 89, 94, 95, 100

[BLM+04] Steffi Beckhaus, Alexander Lechner, Sina Mostafawy, Georg Trogemann, and Richard Wages. **alVRed - Methods and Tools for Storytelling in Virtual Environments**. In *the Internationale Statustagung "Virtuelle und Erweiterte Realität"*, pages 19–20, Leipzig, Germany, 2004. 74, 110

[Blo07a] Kristopher J. Blom. **On Affordances and Agency as Explanatory Factors of Presence**. In *the Extended Abstract Proceedings of the 1st Peach Summer School 2007*. FP6 Coordination Action on Presence, 2007. 93, 220

[Blo07b] Kristopher J. Blom. **Presence of Interactive Expereinces**. Poster at the 1st Peach Summer School, 2007. FP6 Coordination Action on Presence. 44, 93, 220

[BLRS98] Roland Blach, Jürgen Landauer, Angela Rösch, and Andreas Simon. **A Highly Flexible Virtual Reality System**. *Future Generation Computer Systems*, **14**(3-4):167–178, 1998. 106

[BM07] Doug A. Bowman and Ryan P. McMahan. **Virtual Reality: How Much Immersion Is Enough?** *Computer*, **40**(7):36–43, 2007. 93

[boo] **Boost C++ Libraries**. Project website. last visited: Sept 2008. Available from: http://www.boost.org/. 137

[Bow99] Douglas A. Bowman. *Interaction Techniques for Common Tasks in Immersive Virtual Environments Design, Evaluation, and Application*. PhD thesis, Georgia Institute of Technology, June 1999. 20, 48, 89

[BPP95] Gavin Bell, Anthony Parisi, and Mark Pesce. **The Virtual Reality Modeling Language Specification, Version 1.0**. Technical report, 1995. 106

[Bry97] Steve Bryson. **Time, data-time, and real-time interactive visualization**. *Computer Physics*, **11**(3):270–274, 1997. 49

[BSH08] Jacob Benesty, M. Mohan Sondhi, and Yiteng Huang, editors. *Springer Handbook of Speech Processing*. Springer, Berlin, 2008. 60

[BSS03] Andrea Brogni, Mel Slater, and Anthony Steed. **More Breaks Less Presence**. In *PRESENCE 2003, Proceedings of the 6th Annual International Workshop on Presence*, 2003. 92

[BSTH01] Frédéric Boussinot, Jean-Ferdinand Susini, Frédéric Dang Tran, and Laurent Hazard. **A reactive behavior framework for dynamic virtual worlds**. In *Web3D '01: Proceedings of the sixth international conference on 3D Web technology*, pages 69–75, New York, NY, USA, 2001. ACM Press. 74

[Buz91] Tony Buzan. *The Mind Map Book*. Penguin, New York, 1991. 20

[BY03] Johan Bengtsson and Wang Yi. **Timed Automata: Semantics, Algorithms and Tools**. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, **3098** of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003. 33

[Bye07] Celina Byers. **Playing to Learn: Game-Driven Comprehension of Complex Content**. *International Journal of Teaching and Learning in Higher Education*, **19**(1):33–42, 2007. 15

[CAB+00] Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove, Kevin Christiansen, Rob Deline, Jim Durbin, Rich Gossweiler, Shuichi Koga, Chris Long, Beth Mallory, Steve Miale, Kristen Monkaitis, James Patten, Jeff Pierce, Joe Shochet, David Staack, Brian Stearns, Richard Stoakley, Chris Sturgill, John Viega, Jeff White, George Williams, and Randy Pausch. **Alice: Lessons Learned from Building a 3D System For Novices**. In *CHI 2000: Proceedings of the ACM conference on Computer Human Interaction*, pages 486–493. SIGCHI, ACM, 2000. 112

[cal] **Cal3D Character Animation Library**. Project website. last visited: Sept 2008. Available from: http://home.gna.org/cal3d/. 80, 212

[CE01] Antony Courtney and Conal Elliot. **Genuinely Functional User Interfaces**. In *2001 Haskell Workshop*, pages 41–69, September 2001. 243

[CEH03] Jorge Campos, Max J. Egenhofer, and Kathleen Hornsby. **Animation model to Support Exploratory Analysis of Dynamic Environments.** In *Proceedings of Summer Computer Simulation Conference (SCSC)*, pages 761–766, Montreal, Canada, 2003. 2, 113

[CGP99] Edmund M. Clark, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, 1999. 32, 33

[CH93] Christer Carlsson and Olof Hagsand. **DIVE - A Multi-User Virtual Reality System**. In *VRAIS '93: Proceedings of Virtual Reality Annual International Symposium*, pages 394–400. IEEE, 9 1993. 109

[CHE02] Jorge Campos, Kathleen Hornsby, and Max J. Egenhofer. **A Temporal Model of Virtual Reality Objects and their Semantics**. In *DMSA 2002: Eighth International Conference on Distributed Multimedia Systems*, pages 581–588, 2002. 113

[Che06] Yang Chen. **Olfactory Display: Development and Application in Virtual Reality Therapy**. In *ICAT'06: Proceedigns of the 16th International Conference on Artificial Reality and Telexistence–Workshops*, pages 580–584, Los Alamitos, CA, USA, 2006. IEEE Computer Society. 95

[CHLB03] Marc Cavazza, Simon Hartley, Jean-Luc Lugrin, and Mikael Le Bras. **Alternative reality: a new platform for virtual reality art**. In *VRST '03: Proceedings of the ACM symposium on Virtual Reality Software and Technology*, pages 100–107. ACM Press, 2003. 49

[CKP95] James Cremer, Joseph Kearney, and Yiannis Papelis. **HCSM: a framework for behavior and scenario control in virtual environments**. *ACM Transactions on Modelling and Computer Simulation*, **5**(3):242–267, 1995. 2, 16, 111

[CNP03] Antony Courtney, Henrik Nilsson, and John Peterson. **The Yampa Arcade**. In *ACM SIGPLAN Haskell Workshop*, pages 7–18. ACM SIGPLAN, 2003. 150, 227, 243

[Cor91] Daniel D. Corkill. **Blackboard Systems**. *Journal of AI Expert*, **6**(9):40–47, Sept. 1991. 134, 222

[Cor03] Daniel D. Corkill. **Collaborating Software: Blackboard and Multi-Agent Systems & the Future**. In *Proceedings of the International Lisp Conference*, 2003. 134, 222

[Cou04] Antony Courtney. *Modeling User Interfaces in a Functional Language*. PhD thesis, Yale University, May 2004. 150, 154, 183, 226, 227, 233, 236, 239, 243, 245, 248

[CR04] Franck Cassez and Olivier H. Roux. **Structural translation from time Petri nets to timed automata**. In *Proceeding of Fourth International Workshop on Automated Verification of Critical Systems (AVoCS'04)*, Electronic Notes in Theoretical Computer Science. Elsevier Science B. V., 2004. 33

[CS04] Erik Champion and Sachiyo Sekiguchi. **Cultural Learning in Virtual Environments**. In *VSMM2004 Hybrid Realities: Digital Partners*, pages 364–373, 2004. 17

[Del00] Leonidas Deligiannidis. *DLoVe: A specification paradigm for designing distributed VR applications for single or multiple users*. PhD thesis, Tufts University, 2000. 2, 108, 122, 227

[DHP02] Xiangtian Dai, Gregory Hager, and John Peterson. **Specifying Behavior in C++**. In *IRCA'02: the Proceedings of the IEEE International Conference on Robotics and Automation*, pages 153–160, 2002. 226

[DMH+07] Steven Dow, Manish Mehta, Ellie Harmon, Blair MacIntyre, and Michael Mateas. **Presence and engagement in an interactive drama**. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1475–1484, New York, NY, USA, 2007. ACM Press. 76

[Don07] Mary Jo Dondlinger. **Educational Video Game Design: A Review of the Literature**. *Journal of Applied Educational Technology*, **4**(1):21–31, 2007. 15

[DR03] Raimund Dachselt and Enrico Rukzio. **Behavior3D: An XML-Based Framework for 3D Graphics Behavior**. In *Proceedings of the ACM Web3D 2003 Conference*, pages 101–112. ACM Press, 2003. 2, 74, 112

[DV96] Laurent Dami and Didier Vallet. **Higher-Order Functional Composition in Visual Form**. In D. Tsichritzis, editor, *Object Applications*, pages 139–154, 1996. 246, 247, 249

[EH97] Conal Elliott and Paul Hudak. **Functional Reactive Animation**. In *International Conference on Functional Programming*, pages 196–203, 1997. 123, 149, 227, 243

[Ell98] Conal Elliott. **Functional Implementations of Continuous Modeled Animation**. In *Proceedings of PLILP/ALP '98*, pages 284–299, 1998. 123, 243

[Ell99] Conal Elliott. **An Embedded Modeling Language Approach to Interactive 3D and Multimedia Animation**. *IEEE Transactions on Software Engineering*, **25**(3):291–308, 1999. 243

[EN06] Simon Egenfeldt-Nielsen. **Overview of research on the educational use of video games**. *Digital Kompetanse. The Nordic Journal of Digital Literacy.*, **3**:184–213, 2006. 15

[ESYAE94] Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi. **TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications**. *Computer Graphics*, **28**:421–434, 1994. 2, 108, 227, 243

[FGV05] Michael Fischer, Dov Gabbay, and Lluis Vila, editors. *Handbook of Temporal Reasoning in Artificial Intelligence*. Foundations of Artificial Intelligence. Elsevier, 2005. 37, 38

[FKK07] Scott Frees, G. Drew Kessler, and Edwin Kay. **PRISM interaction for enhancing control in immersive virtual environments**. *ACM Transactions on Computer-Human Interaction*, **14**(1):2, 2007. 65

[Gal95] Tinsley A. Galyean. **Guided Navigation of Virtual Environments**. In *Symposium on Interactive 3D Graphics*, pages 103–104, Monterey CA USA, 1995. 77

[GDCV99] Jonas Gomes, Lucia Darsa, Bruno Costa, and Luiz Velho. *Warping & Morphing of Graphical Objects*. Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann, San Francisco, 1999. 46, 81

[Gee06] David Geer. **Vendors Upgrade Their Physics Processing to Improve Gaming**. *Computer*, **39**(8):22–24., Aug. 2006. 122

[Gei98] Christian Geiger. *Schneller Entwurf interaktiver dreidimensionaler Computeranimation*. PhD thesis, Unversity of Paderborn, 1998. 2, 110

[GKKR05] Jeffrey I. Gold, Alexis J. Kant, Seok Hyeon Kim, and Albert Skip Rizzo. **Virtual anesthesia: The use of virtual reality for pain distraction during acute medical interventions**. *Seminars in Anesthesia, Perioperative Medicine and Pain*, **24**(4):203–210, 2005. 11, 16

[GPRR00] Christian Geiger, Volker Palke, Christian Reimann, and Waldemar Rosenbach. **A Framework for the Structured Design of VR/AR Content**. In *VRST '00: Proceedings of the ACM Symposium on Virtual Reality Systems and Technology*, pages 75–82. ACM Press, 2000. 110

[Gre96] Mark Green. **Animation in the Virtual World**. In *Computer Animation*, page 5. IEEE, 1996. 2

[GV03] Claude Girault and Rüdiger Valk. *Petri Nets for Systems Engineering : a guide to modeling, verification and applications*. Springer, 2003. 33

[Han02] Keith Hanna. **Interactive Visual Functional Programming**. In *International Conference on Function Programming (ICFP02)*, 2002. 246

[has] **Haskell.org website**. last checked Oct. 2006. Available from: www.haskell.org. 184, 228

[Has96] Olof Hasgrand. **Interactive Multiuser VEs in the DIVE System**. *IEEE MultiMedia*, **3**(1):30–39, 1996. 109

[Has98] Haskell Language and Library Committee. **Haskell 98 Language and Libraries**. Technical report, 1998. 227, 228

[Has06] Sonja Haselhoff. *Olfaktorisches Display: Einbindung von Gerüchen in interaktive 3D-Welten*. Master's thesis, Universität Hamburg, Germany, 2006. 95

[HB06] Sonja Haselhoff and Steffi Beckhaus. **Benutzerindividuelle, tragbare Geruchsausgabe in Virtuellen Umgebungen**. In *Proceedings of the "Virtuelle und Erweiterte Realität" the 3rd Workshop of the GI working group VR/AR*, pages 83–94, Koblenz, September 2006. Gesellschaft für Informatik, Shaker Verlag. 95

[Hen96] Thomas A. Henzinger. **The Theory of Hybrid Automata**. In *Proceedings of the 11th annual IEEE Symposium on Logic in Computer Science*, pages 278–292. IEEE, 1996. 36

[HH03] Liwen Huang and Paul Hudak. **Dance: A Declarative Language for the Control of Humanoid Robots**. Research Report YALEU/DCS/RR-1253, Yale, 2003. 243

[HMU07] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson/Addison-Wesley, 3rd edition, 2007. 35

[Hof05] Gernot Hoffmann. **Application of Quaternions**. On-line published report, 2005. Last accessed Sep. 2008. Available from: http://www.fho-emden.de/~hoffmann/quater12012002.pdf. 195

[HTW04] A. Helsinger, M. Thome, and T. Wright. **Cougaar: a scalable, distributed multi-agent architecture**. In *IEEE International Conference on Systems, Man and Cybernetics*, **2**, pages 1910–1917, Oct. 2004. 134, 150

[Hua96] Zhiyong Huang. *Motion Control for Human Animation*. PhD thesis, Swiss Federal Institute of Technology, Lausanne (EPFL), 1996. 150

[Hud00] Paul Hudak. *The Haskell School of Expression*. Cambridge University Press, New York, 2000. 228, 243

[Hug00] John Hughes. **Generalising Monads to Arrows**. *Science of Computer Programming*, **37**:67–111, May 2000. 232

[Hug04] John Hughes. **Programming with Arrows**. In *5th International Summer School on Advanced Functional Programming*, **3622** of *LNCS*, pages 73–129. Springer Verlag, 2004. 232

[Hut07] Graham Hutton. *Programming in Haskell*. Cambridge University Press, Cambridge, 2007. 228

[IJs02] Wijnand IJsselsteijn. **Elements of a multi-level theory of presence: Phenomenology, mental processing and neural correlates**. In *Proceedings of PRESENCE conference*, pages 245–259, 2002. 92

[JDM99] Robert J. K. Jacob, Leonidas Deligiannidis, and Stephen Morrison. **A software model and specification language for non-WIMP user interfaces**. *ACM Transactions on Computer-Human Interaction*, **6**(1):1–46, 1999. 108, 227

[JH04] Christopher Johnson and Charles Hansen. *Visualization Handbook*. Academic Press, Inc., Orlando, FL, USA, 2004. 16

[JH07] Jeffrey Jacobsen and Lynn Holden. **Virtual Heritage: Living in the Past**. *Techné: Research in Philosophy and Technology*, **10**(3):55–61, 2007. 17

[JNW06] Dietmar Jackél, Stephan Neunreither, and Friedrich Wagner. *Methoden der Computeranimation*. Springer, Berlin, 2006. 28, 46, 78, 167

[Kel02] Joel Kelso. *A Visual Programming Environment for Functional Languages*. PhD thesis, Murdoch University, 2002. 246, 249, 252, 257

[KEOB04] Merel Krijn, P. M. G. Emmelkamp, Ragnar P. Olafsson, and Roeline Biemond. **Virtual reality exposure therapy of anxiety disorders: A review**. *Clinical Psychology Review*, **24**(3):259–281, July 2004. 16

[KJM03] Bill Kapralos, Michael R. M. Jenkin, and Evangelos Milios. **Auditory Perception and Spatial (3D) Auditory Systems**. Technical Report CS-2003-07, Dept. of Computer Science, York University, 2003. 95, 98

[KS05] Reino Kurki-Suonio. *A Practical Theory of Reactive Systems: Incremental Modeling of Dynamic Behaviors*. Springer-Verlag, 2005. 30, 31

[KSO04] Jared S. Knutzon, Adrian V. Sannier, and James H. Oliver. **An Immersive Approach to Command and Control**. *Journal of Battlefield Technology*, **7**(1):37–42, 2004. 71

[KT99] Marcelo Kallmann and Daniel Thalmann. **Direct 3D Interaction with Smart Objects**. In *VRST '99: Proceedings of the ACM symposium on Virtual Reality Software and Technology*, pages 124–130, New York, NY, USA, 1999. ACM Press. 109

[KYMS06] Pankaj Khanna, Insu Yu, Jesper Mortensen, and Mel Slater. **Presence in response to dynamic visual realism: a preliminary report of an experiment study**. In *VRST '06: Proceedings of the ACM symposium on Virtual Reality Software and Technology*, pages 364–367, New York, NY, USA, 2006. ACM Press. 93

[Lau93] Brenda Laurel. *Computers as Theatre*. Addison-Wesley, Reading, MA, 1993. 89, 92, 93

[LNT00] Kim Gulstrand Larsen, Mogens Nielsen, and P.S. Thiagarajan. **Advanced Tutorial: Timed and Hybrid Automata**. In *"Petri Nets 2000" 21st International Conference on Application and Theory of Petri Nets*. University of Aarhus, June 2000. Tutorial Notes. 33

[Mat02] Micheal Mateas. *Interactive Drama, Art and Artificial Intelligence*. PhD thesis, Carnegie Mellon University, 2002. 76

[MBD05] Wolfgang Minker, Dirk Bhler, and Laila Dybkjaer, editors. *Spoken Multimodal Human-Computer Dialogue in Mobile Environments*, **Vol. 28** of *the Series on Text, Speech and Language Technology*. Springer, Dordrecht, 2005. 48

[MCM99] David L. Martin, Adam J. Cheyer, and Douglas B. Moran. **The Open Agent Architecture: a framework for building distributed software systems**. *Applied Artificial Intelligence*, **13**(1/2):91–128, 1999. 134, 150

[McT04] Michael F. McTear. *Spoken Dialogue Technology: Towards the Conversational User Interface*. Springer-Verlag, London, 2004. 48

[MH06] Benjamin Mesing and Carsten Hellmich. **Using Aspect Oriented Methods to Add Behaviour to X3D Documents**. In *Proceedings of the ACM Web3D 2006 Conference*, Columbia, Maryland, April 2006. ACM. 74, 112

[Mil06] Ian Millington. *Artificial Intelligence for Games*. Morgan Kaufmann, 2006. 46, 206, 207

[MK03] Jixin Ma and Brian Knight. **Representing The Dividing Instant**. *Computer Journal*, **46**(2):213–222, 2003. 25

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, New York, 1992. 30, 31, 32

[MS03] Michael Mateas and Andrew Stern. **Facade: An Experiment in Building a Fully-Realized Interactive Drama**. In *Proceedings of the Game Developer's Conference: Game Design Track*, 2003. 76

[MTT04] Nadia Magnenat-Thalmann and Daniel Thalmann, editors. *Handbook of Virtual Humans*. John Wiley & Sons, Ltd., West Sussex, England, 2004. 44, 110

[Mun05] Mun Hon Cheong. *Functional Programming and 3D Games*. Master's thesis, The University of New South Wales, November 2005. 243

[Mur97] Janet H. Murray. *Hamlet on the Holodeck*. MIT Press, Cambridge, MA, 1997. 2

[NCP02] Henrik Nilsson, Antony Courtney, and John Peterson. **Functional Reactive Programming, continued**. In *Haskell Workshop (Haskell '02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM SIGPLAN, ACM Press. 243

[NNHY06] Fumitaka Nakaizumi, Haruo Noma, Kenichi Hosaka, and Yasuyuki Yanagida. **SpotScents: A Novel Method of Natural Scent Delivery Using Multiple Scent Projectors**. In *VR '06: Proceedings of the IEEE conference on Virtual Reality*, page 29, Washington, DC, USA, 2006. IEEE Computer Society. 95

[NOK+08] Takamichi Nakamoto, Shigeki Otaguro, Masashi Kinoshita, Masahiko Nagahama, Keita Ohinishi, and Taro Ishida. **Cooking Up an Interactive Olfactory Game Display**. *IEEE Computer Graphics and Applications*, **28**(1):75–78, 2008. 95

[Nor02] Donald A. Norman. *The Design of Everyday Things*. Basic Books, New York, 2002. 220

[Pan07] Xueni Pan. **Real Man meets Virtual Woman: A Study of Social Anxiety in a Virtual Environment**. Poster at the 1st Peach Summer School, 2007. 16, 62, 66

[Pap] Dave Pape. **Crayoland**. The homepage of the Crayoland VR environment. last visited: Sept 2008. Available from: http://www.evl.uic.edu/pape/projects/crayoland/. 11, 69

[Pap98] Dave Pape. **Crayoland**. In *SIGGRAPH '98: ACM SIGGRAPH 98 Electronic Art and Animation Catalog*, page 116, New York, NY, USA, 1998. ACM Press. 11, 69

[Par02] Rick Parent. *Computer Animation: Algorithms and Techniques*. Morgan Kaufmann, 2002. 28, 46, 78, 154, 167, 176

[Pat01] Ross Paterson. **A New Notation for Arrows**. In *Proceedings of the International Conference on Functional Programming (ICFP 2001)*. ACM SIGPLAN, ACM Press, 2001. 232, 233, 248

[Pat03] Ross Paterson. **Arrows and Computation**. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 201–222. Palgrave, 2003. 232

[PBC+95] Randy Pausch, Tommy Burnette, A. C. Capehart, Matthew Conway, Dennis Cosgrove, Rob DeLine, Jim Durbin, Rich Gossweiler, Shuichi Koga, and Jeff White. **Alice: Rapid prototyping for virtual reality**. *IEEE Computer Graphics and Applications*, **15**(3):8–11, 1995. 110

[PE08] Mark B. Powers and Paul M.G. Emmelkamp. **Virtual reality exposure therapy for anxiety disorders: A meta-analysis**. *Journal of Anxiety Disorders*, **22**(3):561–569, 2008. 16

[PH03] Izzet Pembeci and Gregory Hager. **Functional Reactive Programming as a Hybrid System Framework**. In *Proceedings of the International Conference on Robotics & Automation*, pages 727–734. IEEE, September 2003. 123, 225, 226

[PHE99] John Peterson, Paul Hudak, and Conal Elliott. **Lambda in Motion: Controlling Robots with Haskell**. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, **1551** of *Lecture Notes In Computer Science*, pages 91–105, 1999. 150, 243

[PHH99] John Peterson, Gregory Hager, and Paul Hudak. **A language for declarative robotic programming**. In *Internation Conference on Robotics and Automation*, **2**, pages 1144–1151, 1999. 243

[PK05] Bernd Page and Wolfgang Kreutzer. *The Java Simulation Handbook*. Shaker Verlag, Aachen, Germany, 2005. 28

[PP06] Wojciech Penczek and Agata Polrola. *Advances in Verification of Time Petri Nets and Timed Automata: A Temporal Logic Approach*. Springer, 2006. 32, 33, 34, 35

[PR02] Lynn Pocock and Judson Rosebush. *The Computer Animator's Technical Handbook*. Morgan Kaufmann, San Francisco, 2002. 28, 167

[PST+96] Randy Pausch, Jon Snoddy, Robert Taylor, Scott Watson, and Eric Haseltine. **Disney's Aladdin: First Steps Towards Storytelling in Virtual Reality**. In *SIGGRAPH '96: Proceedings of the 23th annual conference on Computer Graphics and Interactive Techniques*, pages 193–203, New York, NY, USA, 1996. ACM Press. 2

[PTBK05] Bram Pellens, Olga De Troyer, Wesley Bille, and Frederic Kleinermann. **Conceptual Modeling of Object Behavior in a Virtual Environment**. In *Proceedings of Virtual Concept 2005*, pages 93–94, 2005. 112

[Raj07] Martin Rajman, editor. *Speech and Language Engineering*. EPFL Press, 2007. 66

[Ree95] Hideki John Reekie. *Realtime Signal Processing: Dataflow, Visual, and Functional Programming*. PhD thesis, University of Technology at Sydney, September 1995. 246

[Rey87] Craig W. Reynolds. **Flocks, Herds, and Schools: A Distributed Behavioral Model, in Computer Graphics**. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer Graphics and Interactive Techniques*, **21**, pages 25–34, 1987. 44, 206, 207

[Rey00] Craig Reynolds. **Interaction with Groups of Autonomous Characters**. In *Proceedings of the Game Developers Conference (GDC)*, 2000. 44, 206

[Rhe91] Howard Rheingold. *Virtual Reality*. Secker & Warburg, 1991. 14, 15, 87

[Riz06] Albert "Skip" Rizzo. **Expose, Distract, Motivate and Measure: Virtual Reality Games for Health.** In En J. Snchez, editor, *Nuevas ideas en Informtica Educativa*, **2**, pages 1–4. LOM Ediciones, 2006. 16

[RJL+97] Maria Roussos, Andrew E. Johnson, Jason Leigh, Christina A. Vasilakis, Craig R. Barnes, and Thomas G. Moher. **NICE: combining constructionism, narrative and collaboration in a virtual learning environment**. *Computer Graphics*, **31**(3):62–63, 1997. 49

[RKW01] Sharif Razzaque, Zachariah Kohn, and Mary Whitton. **Redirected Walking**. In *EG '01: Proceedings of Eurographics Conference*, pages 289–294. EuroGraphics Association, 2001. 63

[RLVD04] Gordon D. Rehn, Marco Lemessi, Judy M. Vance, and Denis V. Dorozhkin. **Integrating operations simulation results with an immersive virtual reality environment**. In *WSC '04: Proceedings of the 36th Winter Simulation Conference*, pages 1713–1719, 2004. 49, 109

[RS01] Gerhard Reitmayr and Dieter Schmalstieg. **An Open Software Architecture for Virtual Reality Interaction**. In *VRST '01: Proceedings of the ACM Symposium on Virtual Reality Systems and Technology*, Banff, Alberta, Canada., November 2001. ACM Press. 60, 66, 105

[RWM98] Giuseppe Riva, Brenda K. Wiederhold, and Enrico Molinari, editors. *Virtual Environments In Clincal Psychology and Neuroscience*. IOS Press, 1998. 16

[SC03] William R. Sherman and Alan Craig. *Understanding Virtual Reality: Interface, Application, and Design*. Morgan Kaufmann, San Francisco, CA, USA, 2003. 14, 15, 87, 92, 94, 95, 102

[SCB04] Kenneth Salisbury, Francois Conti, and Federico Barbagli. **Haptic Rendering: Introductory Concepts**. *IEEE Computer Graphics and Applications*, **24**(2):24–32, 2004. 95, 99

[Sch90] Fabio A. Schreiber. **A breviary to time concepts for information systems**. *Rivista di Informatica*, **XX**(1):59–58, 1990. 23

[Sch94] Fabio A. Schreiber. *Real Time Computing*, **F127** of *NATO ASI*, chapter Is Time a Real Time? An Overview of Time Ontology in Informatics, pages 283–307. Springer Verlag, 1994. 22, 23

[Sch98] S. Schiffer. *Visuelle Programmierung - Grundlagen und Einsatzmöglichkeiten*. Addison-Wesley, 1998. 245

[Sch04] Klaus Schneider. *Verification of Reactive Systems*. Springer-Verlag, Berlin Heidelberg, 2004. 30, 31, 32

[SD99] Shamus Smith and David Duke. **Virtual Environments as Hybrid Systems**. In *Proceedings of the Eurographics UK 17th Annual Conference*, pages 113–128. Eurographics UK Chapter, 1999. 113, 114

[SDM01] Shamus Smith, David Duke, and Mieke Massink. **The Hybrid World of Virtual Environments**. *Computer Graphics Forum*, **18**(3):297–308, 2001. 113

[Sho85] Ken Shoemake. **Animating rotation with quaternion curves**. *Computer Graphics*, **19**(3):245–254, 1985. 194

[SK03] Wooyoung Shim and Gerard Jounghyun Kim. **Designing for Presence and Performance: The Case of the Virtual Fish Tank**. *Presence: Teleoperators and Virtual Enviroments*, **12**(4):374–386, 2003. 2

[SKBB08] Roland Schröder-Kroll, Kristopher J. Blom, and Steffi Beckhaus. **Interaction Techniques for Dynamic Virtual Environments**. In *Proceedings of the "Virtuelle und Erweiterte Realität" the 5th Workshop of the GI working group VR/AR*. Gesellschaft für Informatik e.V. (GI), 2008. In Press. 221

[SMC95] Mel Slater, David L. Usoh Martin, and Y. Chrysanthou. **The Influence of Dynamic Shadows on Presence in Immersive Virtual Environments**. In *EGVE '95: Proceedigs of the Eurographics workshop on Virtual Environments*, pages 8–21, 1995. 93

[SML98] Will Schroeder, Kenneth M. Martin, and William E. Lorensen. *The Visualization Toolkit: An Object-oriented Approach to 3D Graphics*. Prentice-Hall, Inc., Upper Saddle River, NJ, 2nd edition, 1998. 16

[SQS04] Manfred R. Schroeder, H. Quast, and H. W. Strube. *Computer Speech: Recognition, Compression, Synthesis (Springer Series in Information Sciences)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2004. 66

[SS00] Mel Slater and Antony Steed. **A Virtual Presence Counter**. *Presence: Teleoperators and Virtual Enviroments*, **9**(5):413–434, 2000. 92

[SS02] Wallace Sadowski and Kay Stanney. *Handbook of Virtual Environments: Design, Implementation, and Applications*, chapter Presence in Virtual Environments, pages 791–796. Lawrence Erlbaum Associates Inc, 2002. 59, 92, 93

[SSLR05] Steffen Strassburger, Thomas Schulze, Marco Lemessi, and Gordon D. Rehn. **Temporally parallel coupling of discrete simulation systems with virtual reality systems**. In *WSC '05: Proceedings of the 37th conference on Winter simulation*, pages 1949–1957. Winter Simulation Conference, 2005. 109

[Ste96] Antony Steed. *Defining Interaction within Immersive Virtual Environments*. PhD thesis, University of London, September 1996. 106

[Ste06] Anthony Steed. **Towards a General Model for Selection in Virtual Environments**. In *proceedings of the IEEE Symposium on 3D User Interfaces (3DUI)*, pages 103–110. IEEE, 2006. 19, 60, 220

[STF00] Jan Springer, Hendrick Tramberend, and Bernd Fröhlich. **On Scripting in Distributed Virtual Environments**. In *4. Immersive Projection Technology Workshop*, Ames, Iowa, June 2000. 108

[Sto97] Oliviero Stock, editor. *Spatial and Temporal Reasoning*. Kluwer Academic Publishers, Dortdrecht, 1997. 37, 38

[Sto06] Herbert Stocker. **Linear Filters - Animating Objects in a Flexible and Pleasing Way**. In *Proceedings of the ACM Web3D 2006 Conference*, pages 119–129, Columbia, Maryland, April 2006. ACM. 112

[Sud06] Thomas A. Sudkamp. *Languages and Machines : An Introduction to the Theory of Computer Science*. Pearson/Addison-Wesley, 2006. 35

[SVS05] Maria V. Sanchez-Vives and B.A. Slater, Mellers. **From Presence to Consciousness through Virtual Reality**. *Nature Reviews Neuroscience*, **6**(4):332–339, 2005. 219

[SZ99] Sandeep Singhal and Michael Zyda. *Networked Virtual Environments. Design and Implementation*. Addison Wesley, Reading, MA., 1999. 47, 100, 222

[Sza07] Piotr Szal. *Visuelle Entwicklungsumgebung zur Erzeugung von Haskell AFRP Code*. Master's thesis, Universität Hamburg, Germany, 2007. 184, 246

[TC95] Patrice Torguet and Rene Caubet. **VIPER - A Virtual Reality Application Design Platform**. In *EGVE '95: Proceedings of the EuroGraphics Symposium on Virtual Environments*. EuroGraphics Association, 1995. 109

[ter95] Alice G.B. ter Meulen. *Representing Time in Natural Language*. MIT Press, 1995. 38

[THS+01] Russell M. Taylor, Thomas C. Hudson, Adam Seeger, Hans Weber, Jeffrey Juliano, and Aron T. Helser. **VRPN: A Device-Independent, Network-Transparent VR Peripheral System**. In *VRST '01: Proceedings of the ACM Symposium on Virtual Reality Systems and Technology*, pages 55–61. ACM Press, 2001. 104

[TJ01] Vildan Tanriverdi and Robert J.K. Jacob. **VRID: a design model and methodology for developing virtual reality interfaces**. In *VRST '01: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 175–182, New York, NY, USA, 2001. ACM Press. 112

[TLG99] Russell Turner, Song Li, and Enrico Gobbetti. **Metis - An Object-Oriented Toolkit for Constructing Virtual Reality Applications.** *Computer Graphics Forum*, **18**(2):121–130, 1999. 108, 122, 227

[Tra99] Henrik Tramberend. **Avocado: A Distributed Virtual Reality Framework**. In *VR'09: Proceedings of IEEE conference on Virtual Reality*, pages 14–21. IEEE Society Press, 1999. 104, 106, 140

[Tra01] Henrik Tramberend. **A Display Device Abstraction for Virtual Reality Applications**. In *Proceedings of the AFRIGRAPH 2001 conference*, pages 75–80. AFRIGRAPH 2001, 2001. 106

[Tra03] Henrik Tramberend. *Avocado: A Distributed Virtual Environment Framework*. PhD thesis, Universität Bielefeld, 2003. 106, 140

[Tur01] Matthew Turk. *Handbook of Virtual Environment Technology*, chapter Gesture Recognition, pages 223–237. Lawrence Erlbaum Associates, Inc, 2001. 66

[vdS00] Peter van der Straaten. *INTERACTION AFFECTING THE SENSE OF PRESENCE IN VIRTUAL REALITY*. Master's thesis, Delft University of Technology, 2000. 93

[Vil03] Hannes Högni Vilhjálmsson. *Avatar Augmented Online Conversation*. PhD thesis, Massachusetts Institute of Technology (MIT), Cambridge, MA, 2003. 62, 66

[Vin03] John Vince, editor. *Handbook of Computer Animation*. Springer, 2003. 28, 167

[Vir] Virtools. Internet Homepage. last visited: Sept 2008. Available from: http://www.virtools.com/. 110

[vP05] Hetty van de Rijt and Frans X. Plooij. *Oje, ich wachse!* William Goldmann Verlag, München, 3rd edition, 2005. German edition of original titled book "Oei, ik groei!". 116

[WB94] Colin Ware and Ravin Balakrishnan. **Reaching for objects in VR displays: lag and frame rate**. *ACM Transactions on Computer-Human Interaction*, **1**(4):331–356, 1994. 59

[WBAS03] Tracy Westeyn, Helene Brashear, Amin Atrash, and Thad Starner. **Georgia Tech Gesture Toolkit: Supporting Experiments in Gesture Recognition**. In *International Conference on Perceptive and Multimodal User Interfaces (ICME 2003)*. ACM, Nov 2003. 66

[WBK01] Andrew Witkin, David Baraff, and Micheal Kass. **Physically based modelling**. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer Graphics and Interactive Techniques*, **Course Notes 25**. ACM SIGGRAPH, 2001. 195

[WBL+96] Robert Welch, Theodore T. Blackmon, Andrew Liu, Barbara A. Mellers, and Lawrence W. Stark. **The effects of pictorial realism, delay of visual feedback, and observer interactivity on the subjective sense of presence**. *Presence: Teleoperators and Virtual Enviroments*, **5**:263–273, 1996. 93

[WBSD+07] Andrea Windich-Biermeier, Isabelle Sjoberg, Juanita Conkin Dale, Debra Eshelman, and Cathie E. Guzzetta. **Effects of Distraction on Pain, Fear, and Distress During Venous Port Access and Venipuncture in Children and Adolescents With Cancer**. *Journal of Pediatric Oncology Nursing*, **24**(1):8–19, 2007. 16

[WELCN97] Terry Welsh, Jeremy Eccles, Uli Lechner, and Carolina Cruz-Neira. **Cueva de Fuego**. Part of Exhibition: "Caveland on Cyberstage" at the CEBIT '97 Festival in Germany., 1997. 68

[WG93] Josie Wernecke and Open Inventor Architecture Group. *The Inventor Mentor: Programming Object-oriented 3D graphics with Open Inventor, release 2*. Addison-Wesley Publishing Company, 2 edition, 1993. 106

[WGW90] Andrew Witkin, Michael Gleicher, and William Welch. **Interactive Dynamics**. In *SI3D '90: Proceedings of the 1990 Symposium on Interactive 3D Graphics*, pages 11–21, New York, NY, USA, 1990. ACM Press. 2, 108, 227

[Whi03] Mary C. Whitton. **Making Virtual Environments Compelling**. *Communications of the ACM*, **46**(7):40–47, 2003. 2, 93, 219

[Why03] Jennifer Whyte. **Industrial applications of virtual reality in architecture and construction**. *ITcon*, **8**:43–50, 2003. 8. Special Issue Virtual Reality Technology in Architecture and Construction. 17

[Wie03] Roel J. Wieringa. *Design Methods for Reactive Systems*. Morgan Kaufmann Publishers, San Fransisco, 2003. 30, 31

[Wil00] Peter Jason Willemsen. *Behavior and Scenario Modeling for Real-Time Virtual Environments*. PhD thesis, The University of Iowa, 2000. 2, 16

[WK88] Andrew Witkin and Michael Kass. **Spacetime constraints**. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer Graphics and Interactive Techniques*, pages 159–168, New York, NY, USA, 1988. ACM. 28

[WP95] Andrew Witkin and Zoran Popovic. **Motion warping**. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 105–108, New York, NY, USA, 1995. ACM. 28

[Zac96] Gabriel Zachmann. **A language for describing behavior of and interaction with virtual worlds**. In *VRST '00: Proceedings of the ACM Symposium on Virtual Reality Systems and Technology*, pages 143–150, Hongkong, July 1996. ACM Press. 2, 112

[Zor03] Diane M. Zorich. **A survey of digital cultural heritage initiatives and their sustainability concerns**. Technical report, D.C. Council on Library and Information Resources., 2003. Retrieved, 11 May 2008. 17

[ZP03] Paul Zimmons and Abigail Panter. **The influence of rendering quality on presence and task performance in a virtual environment**. In *VR '03: Proceedings of IEEE Virtual Reality Conference*, pages 293–294, 2003. 93

[ZPK00] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation: Integrating Discete Event and Continuous Complex Dynamic Systems*. Academic Press, San Diego, 2 edition, 2000. 28, 29, 30